
MA346 Course Notes

Nathan Carter

Jul 29, 2020

CONTENTS

1	Course schedule	3
1.1	Introduction to Data Science	4
1.2	Mathematical Foundations	9
1.3	Jupyter	17
1.4	Review of Python and pandas	22
1.5	Before and After	26
1.6	Single-Table Verbs	32
1.7	Abstraction	38
1.8	Version Control	46
1.9	Mathematics and Statistics in Python	52
1.10	Visualization	62
1.11	Processing the Rows of a DataFrame	88
1.12	Concatenating and Merging DataFrames	106
1.13	Miscellaneous Munging Methods (ETL)	106
1.14	Dashboards	106
1.15	Relations as Graphs - Network Analysis	106
1.16	Relations as Matrices	107
1.17	Introduction to Machine Learning	107
1.18	Big Cheat Sheet	107
1.19	Anaconda Installation	139
1.20	VS Code for Python Installation	139
1.21	GB213 Review in Python	141

These course notes are for [Bentley University](#)'s Data Science course (MA346) that will be taught by [Nathan Carter](#) in Fall 2020.

You can download a PDF of these course notes [here](#), in case you prefer to read it that way.

<p>Warning: This course notes website is still in development, so the content here is incomplete. It will be ready by Fall 2020.</p>

COURSE SCHEDULE

Each topic below will become a link to what we will cover in class that day, once I've made more progress in completing these course notes.

Week	Date	Content for the day
1	9/3/2020	Chapter 1: Introduction to data science (<i>notes</i> , slides)
		Chapter 2: Mathematical foundations (<i>notes</i> , slides)
		DataCamp to review before Week 2
2	9/10/2020	Chapter 3: Computational notebooks (Jupyter) (<i>notes</i> , slides)
		Chapter 4: Python review focusing on pandas and mathematical foundations (<i>notes</i> , no slides)
		DataCamp to review before Week 3
3	9/17/2020	Chapter 5: Before and after, in mathematics and communication (<i>notes</i> , slides)
		Chapter 6: Pandas single-table verbs (<i>notes</i> , slides)
		DataCamp to review before Week 4
4	9/24/2020	Chapter 7: Abstraction in mathematics and computing (<i>notes</i> , slides)
		Chapter 8: Version control and GitHub (<i>notes</i> , slides)
		DataCamp to review before Week 5
5	10/1/2020	Chapter 9: Math and stats in Python (<i>notes</i> , slides)
		Chapter 10: New visualization tools (<i>notes</i> , slides)
		DataCamp to review before Week 6
6	10/8/2020	Chapter 11: Processing the rows of a DataFrame (<i>notes</i> , slides)
		(No DataCamp for next week. Finish project instead.)
7	10/15/2020	Chapter 12: Concatenation and Merging (<i>notes</i> , slides)
		DataCamp to review before Week 8
8	10/22/2020	Chapter 13: Miscellaneous Munging Methods (ETL) (<i>notes</i> , slides)
		DataCamp to review before Week 9
9	10/29/2020	Chapter 14: Dashboards (<i>notes</i> , slides)
10	11/5/2020	Chapter 15: Relations as graphs and network analysis (<i>notes</i> , slides)
11	11/12/2020	Chapter 16: Relations as matrices (<i>notes</i> , slides)

Continued on next page

Table 1.1 – continued from previous page

Week	Date	Content for the day
12	11/19/2020	Chapter 17: Introduction to machine learning (<i>notes</i> , slides)
13	11/26/2020	Thanksgiving break
14	12/3/2020	Final Exam Review
		Final Project Workshop

1.1 Introduction to Data Science

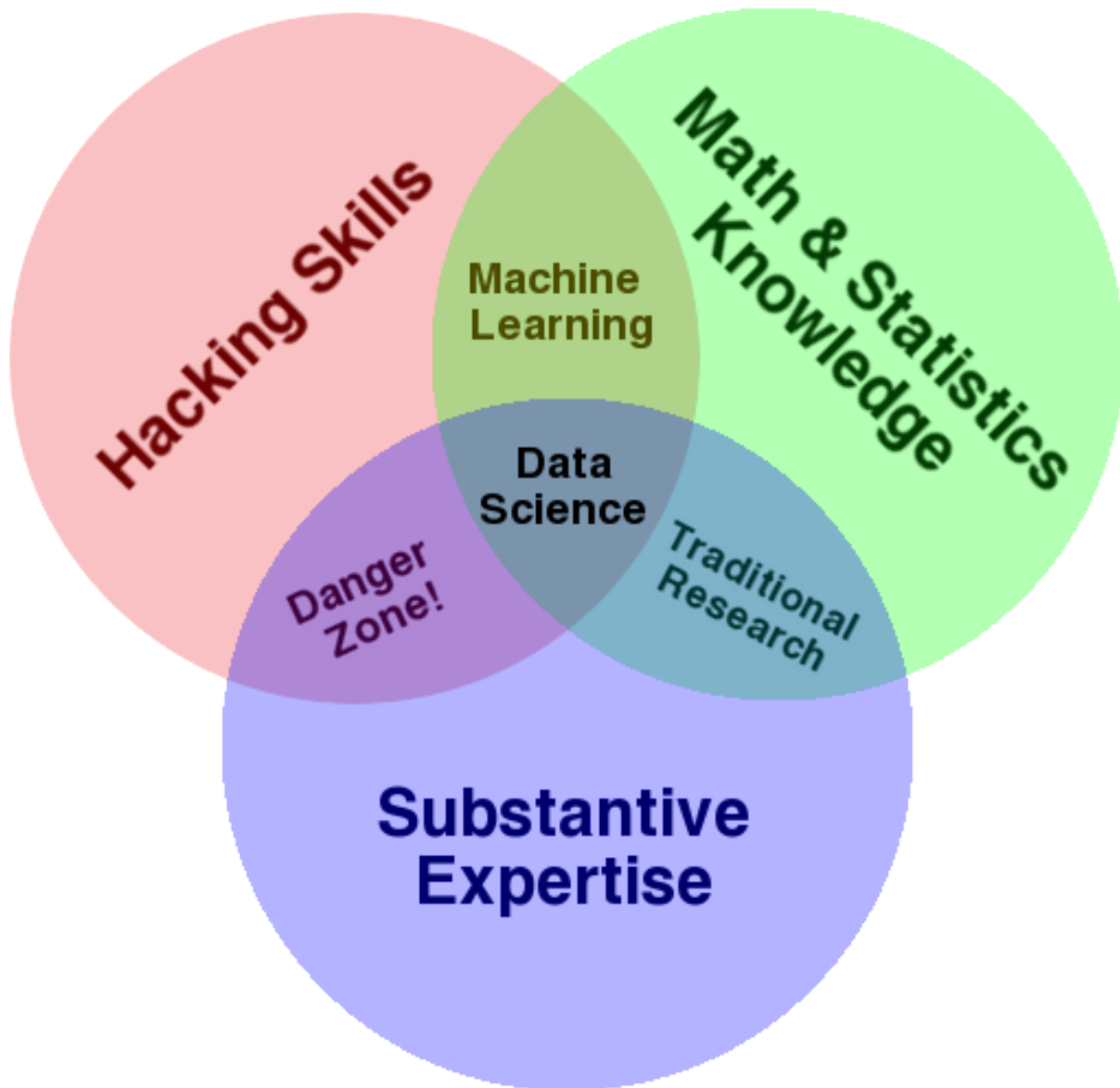
See also the slides that summarize a portion of this content.

1.1.1 What is data science?

The term “data science” was coined in 2001, attempting to describe a new field. Some argue that it’s nothing more than the natural evolution of statistics, and shouldn’t be called a new field at all. But others argue that it’s more interdisciplinary. For example, in *The Data Science Design Manual* (2017), Steven Skiena says the following.

I think of data science as lying at the intersection of computer science, statistics, and substantive application domains. From computer science comes machine learning and high-performance computing technologies for dealing with scale. From statistics comes a long tradition of exploratory data analysis, significance testing, and visualization. From application domains in business and the sciences comes challenges worthy of battle, and evaluation standards to assess when they have been adequately conquered.

This echoes a famous blog post by Drew Conway in 2013, called [The Data Science Venn Diagram](#), in which he drew the following diagram to indicate the various fields that come together to form what we call “data science.”



Regardless of whether data science is just a part of statistics, and regardless of the domain to which we're applying data science, the goal is the same: **to turn data into actionable value**. The professional society INFORMS defines the related field of analytics as “the scientific process of transforming data into insight for making better decisions.”

1.1.2 What do data scientists do?

Turning data into actionable value usually involves answering questions using data. Here's a typical workflow for how that plays out in practice.

1. Obtain data that you hope will help answer the question.
2. Explore the data to understand it.
3. Clean and prepare the data for analysis.
4. Perform analysis, model building, testing, etc.

(The analysis is the step most people think of as data science, but it's just one step! Notice how much more there is that surrounds it.)

5. Draw conclusions from your work.
6. Report those conclusions to the relevant stakeholders.

Our course focuses on all the steps *except for* the analysis. You've learned some introductory statistical analysis in one of the course prerequisites (GB213), and we will leverage that. (Later in our course we will review simple linear regression and hypothesis testing.) If you have taken other relevant courses in statistics, mathematical modeling, econometrics, etc., and want to bring that knowledge in to use in this course, great, but it's not a requirement. Other advanced statistics and modeling courses you take later will essentially plug into step 4 in this data science workflow.

1.1.3 What's in our course?

Our course covers the following four foundational aspects of data science.

- **Mathematics:** We will cover foundational mathematical concepts, such as functions, relations, assumptions, conclusions, and abstraction, so that we can use these concepts to define and understand many aspects of data manipulation. We will also make use of statistics from GB213 (and optionally other statistics courses you may have taken) in course projects, and we will briefly review this material as well. We will also see small previews of other mathematics and statistics courses and their connections to data science, including graphs for social network analysis, matrices for finding themes in relations, and supervised machine learning.
- **Technology:** We will extend your Python knowledge from CS230 with more advanced table manipulation functions, extended practice with data cleaning and manipulation tasks, computational notebooks (such as Jupyter), and GitHub for version control and project publishing.
- **Visualization:** We will learn new types of plots for a wide variety of data types and what you intend to communicate about them. We will also study the general principles that govern when and how to use visualizations and will learn how to build and publish interactive online visualizations (dashboards).
- **Communication:** We will study how to write comments in code, documentation for code, motivations in computational notebooks, interpretation of results in computational notebooks, and technical reports about the results of analyses. We will prioritize clarity, brevity, and knowing the target audience. Many of these same principles will arise when creating presentations or videos as well. Each of these modes of communication is required at some point in our course.

Details about specific topics and their order appears in the course syllabus, and is summarized on [the main page of these course notes](#).

1.1.4 Will this course make me a data scientist?

This course is an introduction to data science. Learning more math, stats, and technology will make you more qualified than just this one course can. (Bentley University has both a [Data Analytics major](#) and a [Data Technologies minor](#), if you're curious which courses are relevant.)

But there are two focuses of our course that will make a big difference.

Learning on your own (LOYO)

Big Picture

I once heard a director of informatics in the health care industry describe how quickly the field of data science changes by saying, “There aren’t any experts; it’s just who’s the fastest learner.” For that reason, it’s essential to cultivate the skill of being able to learn new tools and concepts on your own.

Thus our course requires you to do so. Twice during the course you must partner up with some classmates to research a topic outside of class (from an extensive list the instructor will provide) and report on it to the class, through writing, presenting, video, or whatever modality makes sense for the content.

If you’re interested in a career in this space, I encourage you to follow data scientists on platforms like Twitter and Medium so that you’re kept abreast of the newest innovations and can learn those that are relevant to your work.

Excellent communication

This was already mentioned earlier, but I will re-emphasize it here, because of how important it is. In a meeting between the Bentley University Career Services office and about a dozen employers of our graduates, the employers were asked whether they preferred technical knowledge or what some call “soft skills” and others call “power skills,” which include communication perhaps first and foremost. Unanimously every employer chose the latter.

Big Picture

Data science is about turning data into actionable knowledge. If a data scientist cannot take the results of their analysis and effectively communicate them to decision makers, they have not turned data into actionable knowledge, and have therefore failed at their goal. Even if the insights are brilliant, if they are never shared with those who need them, they achieve nothing. Good communication is essential for data work.

Consequently our course will contain several opportunities for you to exercise your communication skills and receive feedback from the instructor on doing so. See the comments under the “communication” bullet above, and the course outline on [the main page](#). The first such opportunities appear immediately below.

1.1.5 Where should I start?

There are several topics you can investigate on your own that will help you get a leg up in our course. All of these topics are optional, and in our course are available for teams to investigate outside of class and report back, as described above.

Learning on Your Own - File Explorers and Shell Commands

On Windows, the file explorer is called Windows Explorer; on Mac, it is called Finder. It is essential that every computer-literate person knows how to use these tools. Most of the actions you can take with your mouse in Windows Explorer or OS X Finder can also be taken using commands at a command prompt. On Windows, this prompt can be found by running `command.exe`; on Mac, it can be found in `Terminal.app`. It is very useful to know how to do at least basic file manipulation tools with the command prompt, because it enables you to take such action in cloud computing environments where a file explorer is not always available.

A report on file explorers and shell commands would address all of the following points.

- What the folder tree/hierarchy is
 - What a file path is and how to express it on both Windows and OS X
-

- From either the file explorer or command prompt:
 - How to navigate to your home folder
 - How to move up/down the folder hierarchy by one step
 - How to copy or move a file
- From the file explorer:
 - What happens when you double-click a file in a file explorer
 - What file extensions do and when it is acceptable to change them
- From the command prompt:
 - How to list all files in the current folder from the command prompt
 - How to view the contents of a text file

Learning on Your Own - Python IDEs

One of the prerequisites for MA346 is CS230, which introduces the Python language. I assume that you know from that course how to install Python on your own computer and use at least one Python Integrated Development Environment (IDE), such as [PyCharm](#), [VS Code](#), [IDLE](#), [Eclipse \(through PyDev\)](#), [Atom \(through the ide-python package\)](#), and others.

A report on Python IDEs would cover:

- a list of the most commonly used Python IDEs for data science
- very brief installation instructions for each
- a comparison of the major features that distinguish these from one another
- a list of the features that IDEs have that computational notebooks typically do not have
- a demonstration of a few of the most useful distinguishing features of these tools

Learning on Your Own - Numerical Analysis

One valuable contribution that computers make to mathematics is the ability to get excellent approximations to mathematical questions without needing to do extensive by-hand calculations. For instance, recall the trapezoidal rule for estimating the result of an integral (covered in the courses MA126 and MA139). It says that we can estimate the value of $\int_a^b f(x) dx$ by computing the area of a sequence of trapezoids. Choose some points x_0, x_1, \dots, x_n evenly spaced between a and b , with $x_0 = a$ and $x_n = b$, each one a distance of Δx from the previous. Then the integral is approximately equal to $\frac{\Delta x}{2} (f(x_0) + 2f(x_1) + 2f(x_2) + \dots + 2f(x_{n-1}) + f(x_n))$.

A computational notebook reporting on this numerical technique would cover:

- how to implement the trapezoidal rule in Python, given as input some function f , some real numbers a and b , and some positive integer n
- at least one example of how to apply it to a simple mathematical function f where we know the precise answer from calculus, comparing the result for various values of n
- at least one example of how to apply it to a set of data, when a smooth function f is not available

1.2 Mathematical Foundations

See also the slides that summarize a portion of this content.

Big Picture

The contents of this page are extremely foundational to the course. We will be weaving these foundations through every lesson in the course after this one.

1.2.1 Functions

Definition: A *function* is any method for taking a list of inputs and determining the corresponding output.

Examples of functions

Math: We can write functions with the usual notation from an algebra or calculus course:

- $f(x) = x^2 - 5$
- $g(x, y, z) = \frac{x^2 - y^2}{z}$

How is this a method for turning inputs into outputs? Given an input like $x = 2$, a function like f can find an output through the usual mechanism of substitution, more commonly called “plugging it in.” Just substitute 2 into $f(x) = x^2 - 5$ to get $f(2) = 2^2 - 5 = -1$. There are also computer programs into which you can type mathematical notation and ask it to apply the function for you.

English: We can write functions in plain English (or any other natural language, but we’ll use English). To do so, we write a *noun phrase*, and include blanks where the inputs belong:

- the capitol of
- the difference in ages between and

How is this a method for turning inputs into outputs? Given an input like France, I can substitute it into “the capitol of” to get “the capitol of France” and use my knowledge to get Paris. If it were a capitol I didn’t know, I could use the Internet to find out.

Python: We can write functions in Python (or other programming languages, but this course focuses on Python), like this:

```
def square ( x ):
    return x**2

def is_a_long_word ( word ):
    return len(word) > 8
```

How is this a method for turning inputs into outputs? I can ask Python to do it for me!

```
square(50)
```

```
2500
```

```
is_a_long_word( 'Hello' )
```

False

Tables: Any two-column table can work as a function, if we follow a few conventions.

1. The left column will list the possible inputs to the function.
2. The right column will list the corresponding outputs.
3. Each input must show up only once in the table, so there's no ambiguity about what its corresponding output is.

Here's an example, which converts Bentley email IDs to real names for a few members of the Mathematical Sciences Department:

User ID	Name
aaltidor	Alina Altidor
mbhaduri	Moinak Bhaduri
wbuckley	Winston Buckley
ncarter	Nathan Carter
lcherven	Luke Cherven

(We could add more names, but it's just an example.)

How is this a method for turning inputs into outputs? We use the familiar and fundamental operation of *lookup*, something that shows up in numerous places when working with data. (We'll return to the concept of lookup at the end of this chapter.) Given a User ID as input, we look for it in the first column of the table, and once it's found, the appropriate output is right next to it in the right column.

Others: Later in the course we will see other ways to represent functions, but the ones above are the most common.

Which way is best?

The examples above show that you can express functions using math, English, Python, tables, and more. Although none of these ways is always better than the others, we will typically give functions names and refer to them by those names. Examples:

- In Math: Rather than writing out $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ all the time, people just use the short name “the quadratic formula.”
- In Python: The `def` keyword in Python is for giving names to functions so that you can use them later by just typing their name.

Why care about functions?

The concept of a function was invented because it represents an important component of how humans think about the processing of information. As you've seen above, functions show up in ordinary language, in mathematics, in tables of data, and code that processes data. Even people who don't do data work use functions unknowingly all the time when they talk about information, as in:

- I don't know all the state capitols. (In other words, I haven't memorized the function that gives the capitol for a state.)
- You better learn your times tables. (In other words, you should memorize the function that gives the product of two small whole numbers.)
- What's Kayla's phone number? (In other words, please apply the phone-number-of-person function to Kayla for me.)

Unsurprisingly, functions show up all over the place in data science. In particular, when working with a pandas DataFrame, we use functions often to summarize columns (such as compute the max, min, or mean) or to compute new columns, as in this example using Python's built in `/` function:

```
df['Per capita cost'] = df['Cost'] / df['Population']
```

1.2.2 Writing functions in Python

In MA346, we'll almost always want the functions we use to be written in Python, so that we can run them on data. Let's practice writing some functions in Python.

Exercise 1 - from mathematics

Write a function `solve_quadratic` that takes as input three real numbers a , b , and c , and gives as output a list of all real number solutions to the equation $ax^2 + bx + c = 0$. (It can return an empty list if there are no real number solutions.)

Example: `solve_quadratic(1, 0, -4)` would yield `[-2, 2]` because $1x^2 + 0x + (-4) = 0$ is the same equation as $x^2 = 4$.

The above exercise requires only the basic arithmetic built into Python, but when we do more advanced mathematics and statistics, we will import tools like `numpy` and `scipy`.

Exercise 2 - from English

Write a function `last_closing_price` that takes as input a NYSE ticker symbol and gives as output the price of one share at the last closing time of the NYSE. Hints:

- The URL <https://finance.yahoo.com/quote/GOOG> gives data for Alphabet, Inc. A similar URL works for any ticker symbol.
- You can extract all tables from a web page as pandas DataFrames as follows:

```
data_frames = pd.read_html( 'put the URL here' )
```

- That page has only one table, so it will be `data_frames[0]`.

Example: `last_closing_price('GOOG')` yielded something like `1465.85` in mid-June 2020.

It is not always guaranteed that you can turn an idea expressed in English, like “look up the last closing price of a stock,” into Python code. For instance, no one knows how to write code that answers the question, “given a digital photo as input, return the year in which the photo was taken.” But coming up with creative ways to answer important questions in code is a very valuable skill we will work to develop.

Exercise 3 - from a table

Write a function `country_capitol` that takes as input a string containing a country name and gives as output a string containing the name of the country's capitol. Hints:

- A list of countries and capitols appears here: <https://www.boldtuesday.com/pages/alphabetical-list-of-all-countries-and-capitals-shown-on-list-of-countries-poster>
- To convert two columns of a pandas DataFrame into a Python dict for easy lookup, try the following.

```
D = dict( zip( df['input column name'], df['output column name'] ) )
```

- You can then look items up using `D[item_to_look_up]`, as in `D['ZIMBABWE']`.

Example: `country_capitol('JORDAN')` would yield `'AMMAN'`.

Why do you think the `dict(zip())` trick given above works? What exactly is it doing?

1.2.3 Terminology

The following terminology is used throughout computing when discussing functions.

Definition: A *data type* is a category of values.

For instance, `int` is a Python data type for integers (that is, positive and negative whole numbers). Each number is a value in that data type. Other Python data types include `bool` (with the values `True` and `False`), `str` (short for “string” and containing text), and more.

Definition: A function’s *input type* is the type of values you can pass as inputs when calling the function. If a function has multiple inputs, we might speak of its *input types* instead.

In Python, we are not required to write the input types of functions into our code, so we can only know them by reading a function’s documentation or by inspecting the function’s code and reasoning it out.

For example, the `square` function defined above probably has input type `float` (any number). The `is_a_long_word` function has input type `str`.

Definition: A function’s *output type* is the type of values the function returns as outputs. Not all functions have a single return type, but many do.

For example, the `square` function always produces a `float` output and the `is_a_long_word` function always produces a `bool` output.

These ideas of input type and output type are a bit related to the ideas of domain and range of functions in mathematics, but they are not precisely the same. The difference is not important here.

Definition: A function is sometimes called a *map* from its input type to its output type. We say that a function *maps* its inputs to its outputs.

For instance, the `is_a_long_word` function maps strings to booleans.

Definition: A function that takes a single input is called a *unary* function. If it takes two inputs, it is a *binary* function. If it takes three inputs it is a *ternary* function. The number of inputs is called the *arity* of the function.

Although there are related words that go beyond three inputs (quaternary!) almost nobody uses them; instead, we would probably just say “a four-parameter function.”

1.2.4 Relations

Definition: A *relation* is a function whose output type is `bool`, that is, the outputs are always either true or false.

Examples of relations

Math: Any equation or inequality in mathematics is a relation, such as $x^2 + y^2 \geq z^2$ or $x \geq 0$.

Consider $x \geq 0$. Given any input of the appropriate type, say $x = 15$, we can determine a true or false value by substitution. In this case, substituting $x = 15$ into $x \geq 0$ gives $15 \geq 0$, which we know is true. We could do a similar thing with $x^2 + y^2 \geq z^2$ if given three numerical inputs instead of just one.

English: Any declarative sentence with blanks in it is a relation, such as “ is the capitol of ” or “ is a fruit.”

Given any input, you can use it to fill in the blank in the sentence and then judge (using your ordinary knowledge of the world and English) whether the sentence is true. For instance, if we’re working with the sentence “ is a fruit” and I provide the input “Python,” then I get the sentence “Python is a fruit,” which is obviously false, because it’s a programming language, not a fruit.

Python: Any Python function with output type `bool` is a relation.

You can evaluate such relations by running them in Python, just as we did with functions earlier. In fact, the `is_a_long_word` function from earlier is not only a function, but also a relation. Here are two other examples:

```
def R ( a, b ):
    return a in b[1:]

def is_a_primary_color ( c ):
    return c in ['red', 'green', 'blue']
```

Although the first relation is an example with no clear purpose, the second one has a clear meaning. We can test it out like so:

```
is_a_primary_color( 'blue' ), is_a_primary_color( 'orange' )
```

```
(True, False)
```

Lists: A very common way of defining a relation is to just list all the inputs for which the relation is true, and then we know that everything else makes it false.

In data science, we often do this using tables. For example, consider the table on the webpage mentioned in Exercise 3, above. That table lists all the pairs of inputs that make the “ is the capitol of ” relation true.

If you want to check whether, for example, “Bangalore is the capitol of India” is true, you can look to see if any row of the table is `('India', 'Bangalore')`. Since there is no such row, the relation is false for that input. (The capitol is actually New Delhi.)

Big Picture

Every table is a relation. Each row represents a set of inputs that would make the relation true, and any inputs that don’t appear as a row in the table make it false.

Thus every pandas DataFrame is a relation, every SQL table is a relation, and every table you see printed in a book or on a webpage is a relation. This is why SQL is the language for querying *relational* databases.

The above big picture concept is almost 100% true. Technically, a pandas DataFrame or an SQL table can have repeated rows, which is unnecessary if you’re defining a relation. And technically pandas DataFrames and SQL tables also have an extra layer of data called the “index” which we’re ignoring for now, just concentrating on the contents of the table’s columns.

Others: Later in the class we’ll see even other ways to represent functions.

Which way is best?

Although we can express relations in all the ways just mentioned—in math, English, Python, or with lists—we typically *talk about* relations by using simple phrases. For instance, it’s awkward to say “the ‘`is a fruit`’ relation,” so I would probably instead say something like “being a fruit.” And instead of $x < y$, I might say something like “the usual less-than relation for numbers.”

Sometimes we just use the central phrase to describe a binary relation. So to discuss the “`has more employees than`” relation, I might just use the phrase “has more employees than” when talking about it, or perhaps just “more employees.” Usually it’s clear what we mean.

Why care about relations?

The mathematical concept of a relation was invented because humans use it all the time when we think and speak, even though we don’t precisely define it in everyday life. Every time we say a declarative sentence, this idea comes up. Here are some examples:

- If I say, “George isn’t friends with Mia,” then I’m relying on your familiarity with the being-friends-with relation, which you’ve known since Kindergarten.
- If I say, “Dell acquired EMC in 2015,” then I’m relying on your familiarity with the “acquired” relation among companies, which you might not have been very familiar with before coming to Bentley.

The above examples are from binary relations, which are possibly the most common type. Just as a function can be binary (that is, take two inputs), so can a relation, because it’s just a special type of function. But of course we can have unary functions as well (taking one input only), like the `is_a_long_word` and `is_a_primary_color` examples above, and we can have relations with three or more inputs as well.

A very important use of relations in data science is for *filtering* a dataset. We often want to focus our attention on just the section of a dataset we’re interested in, which we describe as “filtering” to keep the rows we want (or “filtering out” the rows we don’t want). In pandas, you can select a subset of a DataFrame `df` and return it as a new DataFrame (or, rather, a view on the original), like so:

```
# To filter the rows of a DataFrame, index the DataFrame with the relation:
df[put_any_relation_here]

# Here's an example, which uses the >= relation to filter for adults:
df[df['age'] >= 18]
```

1.2.5 Relations and functions in data

Exercise 4 - food inspections

The table below shows a sample of data taken from a [larger dataset on data.world](#) about Chicago city food inspections. Imagine the entire dataset of over 150,000 rows based on the sample of the first 10 rows shown below.

1. Name at least two relations expressed by the contents of this table. (You need not use all the columns.)
 2. What are the input types, output type, and arity of each of your relations?
 3. Does the table contain any sets of columns that define a function?
 4. If so, what are the input types, output type, and arity of the function(s)?
-

Business	Address	Inspection Date	Inspection Type	Results
ZAM ZAM MIDDLE EASTERN GRILL	3461 N CLARK ST	11/07/2017	Complaint	Pass
SPINZER RESTAURANT	2331 W DEVON AVE	11/07/2017	Complaint Re-Inspection	Pass
THAI THANK YOU RICE & NOODLES	3248 N LINCOLN AVE	11/07/2017	License Re-Inspection	Pass
SOUTH OF THE BORDER	1416 W MORSE AVE	11/07/2017	License	Pass
BEAVERS COFFEE & DONUTS	131 N CLINTON ST	11/07/2017	License	Not Ready
BEAVERS COFFEE & DONUTS	131 N CLINTON ST	11/07/2017	License	Not Ready
BEAVERS COFFEE & DONUTS	131 N CLINTON ST	11/07/2017	License	Not Ready
FAT CAT	4840 N BROAD-WAY	11/07/2017	Complaint Re-Inspection	Pass
SAFARI SOMALI CUISINE	6319 N RIDGE AVE	11/07/2017	License	Fail
DATA RESTAURANT	2306 W DEVON AVE	11/06/2017	Complaint	Out of Business

Exercise 5 - tech companies

The table below shows a sample of data taken from a [larger dataset on data.world about the 2016 Technology Fast 500](#). Imagine the entire dataset of 500 rows based on the sample of the first 10 rows shown below.

1. Name at least two relations expressed by the contents of this table. (You need not use all the columns.)
2. What are the input types, output type, and arity of each of your relations?
3. Does the table contain any sets of columns that define a function?
4. If so, what are the input types, output type, and arity of the function(s)?

CEO Name	City	Company Name	Country	Market	State
Charles Deguire	Boisbriand	Kinova Inc.	Canada	Canada	QC
Greg Malpass	Burnaby	Traction on Demand	Canada	Canada	BC
Jack Newton	Burnaby	Clio	Canada	Canada	BC
Jory Lamb	Calgary	VistaVu Solutions Inc.	Canada	Canada	AB
Wayne Sim	Calgary	Enersight	Canada	Canada	AB
Bryan de Lottinville	Calgary	Benevity, Inc.	Canada	Canada	AB
J. Paul Haynes	Cambridge	eSentire	Canada	Canada	ON
Jason Flick	Kanata	You.i TV	Canada	Canada	ON
Matthew Rendall	Kitchener	Clearpath	Canada	Canada	ON
Dan Latendre	Kitchener	Igloo Software	Canada	Canada	ON

1.2.6 Some technical notes

Connections between functions and relations

As you've probably noticed, there are some close relationships between relations and functions. Let's state them explicitly.

- Our definitions say that a relation is *a special kind of function*; that is, it's one whose output type has to be bool. So every relation is really also a function.
- But in the last two exercises, we've been thinking about relations and functions in tables. There we saw that we can think of a function as *a special kind of relation*; that is, it's one in which one column has all unique values, so that it can be used for input lookup in an unambiguous way.

Applying functions and relations

This idea of “input lookup” is called *applying* a function. For example, we apply the `country_capitol` function by looking up the country in the table and giving the corresponding capitol as output.

But we can actually do lookup in a relation as well, as long as we don't mind the possibility of getting more than one output. For instance, if we use the Technology Fast 500 table shown above and look up a city name, and ask for the corresponding company name, we won't always get just one answer. Even in just the small sample of the data we have, we can see that Calgary houses at least three different companies.

In short, functions let you apply them and get a unique answer, while relations let you apply them and get any number of answers.

Inverses

As mentioned above, a function is a relation in which *for each input, there is exactly one output*. But for *some* functions, the reverse is also true: For each *output*, there is exactly one *input*.

For example, consider the Technology Fast 500 table again, and let's assume that each company and CEO name is unique (i.e., there are not two CEOs name Jack Newton, or two companies named Clearpath, etc.). Consider the function that maps a company name to the corresponding CEO name; let's call it `find_ceo_for_company`.

- As with every function, for each input company, there is exactly one CEO output.
- But in this case, also, for each CEO output, there is exactly one input company.

While we chose to use the company as input and provide the CEO name as output, we could also have done it in the other order. That is, we could have created a function `find_company_for_ceo` that takes a CEO name as input and provides the corresponding company name as output. It just depends on which column you chose to use as the input and which you choose to use as the output.

This concept is probably familiar from mathematics, where we speak of *inverting* a function. In mathematical notation, we write the inverse of f as f^{-1} , but in computing, we can use more descriptive names, like the example of `find_ceo_for_company` and `find_company_for_ceo`.

In summary: For a relation to be a function, it has to provide just one output for each input. For it to be invertible, it has to have just one input for each output.

1.2.7 An extremely common data operation: Lookup

When working with data and writing code, we “look up” values in many different ways. We’ve already discussed above how applying a function expressed in a table is done by looking up the input and finding the corresponding output.

Let’s review the most common ways that lookup operations show up in Python coding. Almost all of them use square brackets, because that’s the common coding notation for looking up an item in a larger structure.

1. If we have a Python list `L` then we can look up the fourth item in it using the syntax `L[3]`, for example. In this way, you can think of a list as a function from numbers to the contents of the list.
2. If we have a Python dictionary `D` then we can look up an item in it using the syntax `D[my_item]`. So a dictionary is very much like a function; it maps its keys to their corresponding values.
3. If we have a pandas DataFrame, there are many ways to look up items in it, including:
 - filtering for just some rows, as discussed earlier, using syntax like `df[df.X==Y]`, and then selecting the column to use as the result, as in `df[df.Name=='Smith'].Employer`
 - choosing one or more rows and/or columns by their names, using `df.loc[rows,cols]`, as in `df.loc['May':'June','Rainfall']`
 - choosing one or more rows and/or columns by their zero-based index, using `df.iloc[rows,cols]`, as in `df.iloc[:,5]`

Some of the lookup operations shown above act like functions and some act like relations. For instance, a Python list always returns one value when you use square brackets for lookup, so that behaves like a function. But a pandas DataFrame might yield multiple values when you execute code like `df[df.Name=='Smith'].Employer`, because there may be many Smiths in the dataset. If you don’t care about getting *all* the results, but want to just choose one of them, you can always add `.iloc[0]` on the end of the code to select just the first result from the list, as in `df[df.Name=='Smith'].Employer.iloc[0]`.

Later in the course we will see that SQL joins (called by various names in pandas, including merge, concat, and join) are highly related to all the lookup concepts just discussed. A SQL or pandas join is like doing many lookups all at once, which is why it is such a common operation.

1.3 Jupyter

See also the slides that summarize a portion of this content.

1.3.1 What’s Jupyter?

The Jupyter project makes it possible to use code to experiment with and process data in your web browser. It lets you do all of these things in one page (or browser tab):

- write and run code
- write explanations of code and data, including with mathematical formulas
- view tables, plots, and other visualizations of data
- interact with certain types of data visualizations

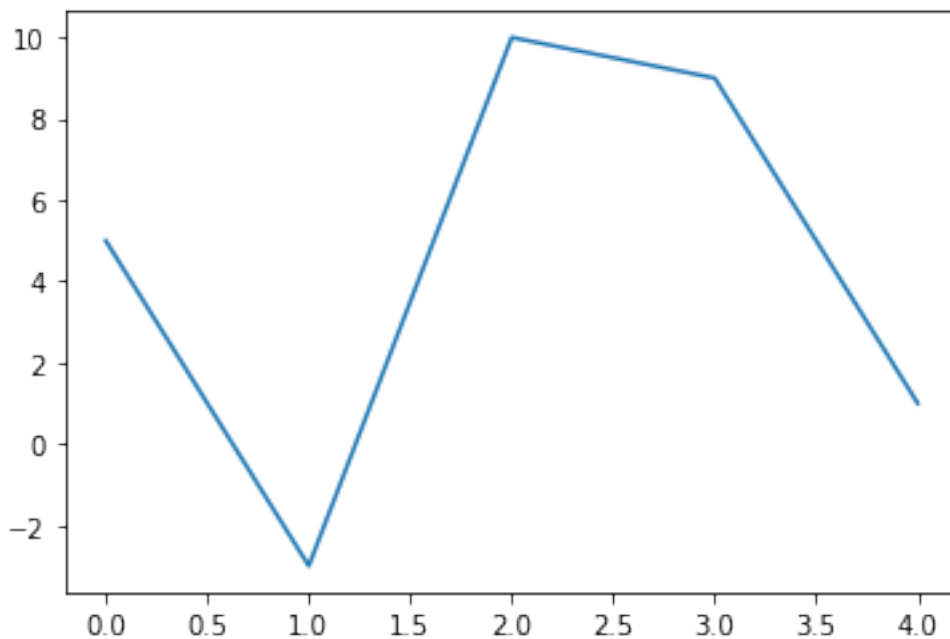
It’s pronounced just like Jupiter, but has the funny spelling because it was originally built for Python, so they wanted to work a “py” in there somewhere.

You may prefer to use another tool to accomplish these tasks; your MA346 instructor won’t force you to use Jupyter. But you should still know about Jupyter for the following reasons:

- Lots of people in data science and analytics use Jupyter notebooks, so you'll definitely encounter them and want to be familiar with how to read them, edit them, and run them.
- It's becoming the *lingua franca* for how to share your data research online, so you may want to know how to publish Jupyter notebooks, something our course will cover.
- It was a big enough deal to win one of the highest awards in the computer science profession, the [2017 ACM Software System Award](#).

In fact, these course notes were written in Jupyter. That's why you'll see code inputs and outputs interspersed among them, because Jupyter lets you write documents with code built in, and it runs the code for you and shows you the output. Here's an example:

```
import matplotlib.pyplot as plt
plt.plot( [5, -3, 10, 9, 1] )
plt.show()
```



Okay, sounds great, so where do we point our browser to start using this thing? Well, you've got lots of options, so let's see what they are first.

1.3.2 How does Jupyter work?

Big Picture

Jupyter is made of two pieces:

1. The notebook interface, which shows you a document with code and visualizations in it, called “a Jupyter notebook.”
2. The engine behind the notebook, which runs your code, and is doing its work invisibly in the background; this engine is called the “kernel.”

How you interact with each of these two pieces is important, and comes with some pitfalls to avoid.

Jupyter in the cloud

The easiest way to start using Jupyter is to just point your browser at a website that offers you access to Jupyter in the cloud. In such a situation, Jupyter's two pieces work like this:

1. The notebook interface runs in your browser on your computer
2. The kernel runs in the cloud on a server provided by someone else

Here are three examples of where you can use Jupyter notebooks in the cloud:

1. Best choice: [Deepnote](#)
 - They took the standard Jupyter interface and added more goodies
 - You can read and write files to/from your Google Drive
 - You can use multiple languages (though we'll stick to Python)
 - The amount of computing power they give you for free is pretty good
 - It's extremely easy to share your work with anyone
 - But you have access to Deepnote only because I signed our class up; they're a startup and they're not letting everyone in yet
2. Next best choice: [Google Colab](#)
 - All the same positives as Deepnote except without as many interface goodies
 - Supports more languages than Deepnote does, but that's not relevant in MA346
3. Third choice: [CoCalc](#)
 - Many features that the previous two don't have, including a nice palette of common code snippets you can insert
 - But the notebook interface is nonstandard and different from Jupyter's in several ways
 - Perhaps the most limited in terms of how much computing you get for free, though this is not very important for MA346

Obviously, none of these is going to give you access to a supercomputer for free. If you want to do any intense or lengthy computing in the cloud, you have to pay for them to let the kernel you're using run on big hardware.

Jupyter on your machine

You can also choose to run Jupyter on your own machine. In contrast to accessing Jupyter in the cloud, when you run it on your own machine, Jupyter's two pieces work like this:

1. The notebook interface still runs in your browser on your computer
2. The kernel now also runs on your computer, which has both advantages and disadvantages

Let's consider the major tradeoffs in each of these approaches.

Why put Jupyter on my computer?	Why choose the cloud instead?
1. Not limited by how much power a cloud company will give you for free.	1. You don't have to install anything on your computer.
2. Even if I don't have good wifi access, I can still use it.	2. You can use it on a phone/tablet.
3. May be easier to add specific Python packages you need for your work.	3. Avoid accidentally leaving a kernel running invisibly. (See below.)

If you want to go this route, there are several ways to install Jupyter on your machine.

Easiest way: *Install Anaconda*

- This is by far the easiest method, so start here.
- Follow the link above for detailed instructions within these course notes; it is not necessary to also install VS Code, which the instructions make optional.
- Once Anaconda is installed, you can launch it from the Windows Start menu or Mac Applications folder, then choose to launch either Jupyter Lab or the Jupyter Notebook.

Before we discuss the other methods of installing Jupyter, let's discuss the difference between Jupyter Lab and the Jupyter Notebook. Here's a summary:

Jupyter Notebook	Jupyter Lab
The original Jupyter project	Its newer successor
Uses multiple browser tabs	Does everything in one tab
Supports many extensions	Doesn't yet support all extensions
Has no console/terminal access	Has both console and terminal access

Both technologies let you edit Jupyter notebooks. (Yes, it's confusing that one app is called "the Jupyter Notebook" and the files are also called "Jupyter notebooks." Sorry.)

Big Picture

When you launch either the Jupyter Notebook or Jupyter Lab, you launch both the user interface (which you see in your browser) and the kernel (which you don't!). **Just closing the browser tab DOES NOT CLOSE THE KERNEL.** Doing this repeatedly (e.g., each day in class) will clog up your computer with many kernels running invisibly in the background.

Instead, do one of these things EVERY TIME you're done coding:

- In Jupyter Notebook: File Menu > Close and Halt
- In Jupyter Lab: File Menu > Shut down

These close the (invisible) kernel first, then let you close the user interface after that.

But that's a hassle! Wouldn't it be easier if Jupyter were just an app I could run on my machine, like every other app? In fact, because Jupyter is *not* an app, you can't even double-click Jupyter notebook files (which end with the `.ipynb` extension) and have them automatically open in Jupyter. Another hassle! How can we fix these things?

Another option: Install *nteract* (pronounced "interact")

- This assumes you have a working Python installation. The easiest way to do that is to install Anaconda, using the instructions up above. That's why this one is listed second; it assumes you've done that first.
- Then visit the website linked to above and follow the very easy process of installing the nteract app.
- When you run nteract, it shows you a new, blank Jupyter notebook. It has already launched the invisible kernel behind the scenes for you. (No need to go to Anaconda Navigator first!)
- You can also double-click notebooks to open them in nteract. Easy, just like every other app on your machine.
- When you quit nteract, it quits not only the user interface you see, but the invisible kernel as well. Nothing to remember.

The only disadvantage here is that some Jupyter notebook extensions don't work in nteract. But we won't be using many of those in MA346 anyway.

1.3.3 Closing comments

There are many websites that make it easy to view Jupyter notebooks online. This is very useful for sharing the results of your work when you're done. Examples include [NbViewer](#) and [GitHub](#), but there are others. Notebooks are often shared in nerdy places on the Internet, with websites supporting viewing them with all their plots, tables, and math displayed nicely. We will learn how to use GitHub in a future week.

There are various pros and cons to using Jupyter notebooks vs. plain old Python scripts, as you probably did in CS230. There are also some hybrid technologies that exist to make notebooks more like scripts, or scripts more like notebooks (such as [Papermill](#), [VSCoDe notebook support](#), and others). In this class, you can usually use whatever technology you prefer. The instructor will use notebooks because they are good for communicating, and communicating is your instructor's job.

Learning on Your Own - Problems with Notebooks

Some folks [really don't like Jupyter notebooks](#). And they have good points! Study what pitfalls notebooks have, based on the presentation at that link, and report on them to the class.

Such a report would include:

- From the many problems the presentation lists, choose the 4-6 that are most relevant to MA346 students.
- For each such problem:
 - Explain it carefully.
 - Show how a tool other than Jupyter doesn't have the same problem.
 - Suggest specific ways that MA346 students can avoid pitfalls surrounding that problem.

Learning on Your Own - Math in Notebooks

You can add mathematics to Jupyter notebooks and it looks very nice. Here's an example of the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

This can be useful for explaining mathematical and statistical concepts in your work clearly, without resorting to ugly text attempts to look sort of like math.

Such a report would include:

- An explanation of what a student would type into a Markdown cell to make some simple mathematics
- A list of the 5-10 most common math notation or symbols the student will want to know how to create (particularly those relevant to statistics and/or data science)
- Suggestions for where the student can go to learn more symbols or notation if they need it

1.4 Review of Python and pandas

Unlike most chapters, there are no slides corresponding to this chapter, because they consist mostly of in-class exercises. They aim to help you remember the Python and pandas you learned in CS230 and be sure they're refreshed and at the front of your mind, so that we can build on them in future weeks.

1.4.1 Python review 1: Remembering pandas

This first set of exercises works with a database from the U.S. Consumer Financial Protection Bureau. The dataset recorded all mortgage applications in the U.S. in 2018, over 15 million of them. Here we will work with a sample of about 0.1% of that data, just over 15 thousand entries. These 15 thousand entries are randomly sampled from just those applications that were for a conventional loan for a typical home purchase of a principal residence (i.e., not a rental property, not an office building, etc., just standard house or condo for an individual or family).

Download the dataset as a CSV file [here](#). If you have questions about the meanings of any column in the dataset, they are fully documented [on the government website](#) from which I got the original (much larger) dataset.

Exercise 1

In class, we will work independently to perform the following tasks, using a cloud Jupyter provider such as Deepnote or Colab.

1. Create a new project and name it something sensible, such as “MA346 Practice Project 1.”
2. Upload the data file into the project.
3. Start a new Jupyter notebook in the same project and load the data file into a pandas DataFrame in that notebook.
4. Explore the data using pandas’s built-in `info` and/or `head` methods.
5. The dataset has many columns we won’t use. Drop all columns except for `interest_rate`, `property_value`, `state_code`, `tract_minority_population_percent`, `derived_race`, `derived_sex`, and `applicant_age`.
6. Reading a CSV file does not always ensure that columns are assigned the correct data type. Use pandas’s built-in `astype` function to correct any columns that have the wrong data type.
7. Practice selecting just a subset of the DataFrame by trying each of these things:
 - Define a new variable `women` that contains just the rows of the dataset containing mortgage applications from females. How many are there? What are the mean and median loan amounts for that group?
 - Repeat the previous bullet point, but for Asian applicants, stored in a variable named `asians`.
 - Repeat the previous bullet point, but for applicants whose age is 75 or over, stored in a variable `age75andup`.
8. Make your notebook presentable, using appropriate Markdown comments between cells to explain your code. (Chapter 5 will cover best practices for how to write such comments, but do what you think is best for now.)
9. Use Deepnote or Colab’s publishing feature to create a shareable link to your notebook. Paste that link into our class’s Microsoft Teams chat, so that we can share our work with one another and learn from each other’s work.

Learning on Your Own - Basic pandas work in Excel

Investigate the following questions. A report on this topic would give complete answers to each.

- Which of the tasks in Exercise 1 are possible to do in Excel and which are not?
- For those that are possible in Excel, what steps does the user take to do them?

- Will the resulting Excel workbook continue to function correctly if the original data changes?
 - Which steps are more convenient in Excel and which are more convenient in Python and pandas, and why?
-

1.4.2 Adding a new column

As you may recall from CS230, you can add new columns to a pandas DataFrame using code like the example below. This example calculates how much interest the loan would accrue in the first year. (This is not fully accurate, since of course the borrower would make some payments that year, but it's just an example.)

```
df['interest_first_year'] = df['property_value'] * df['interest_rate'] / 100
df.head()
```

Running this code in the notebook you've created would work just fine, and would create that new column. It would have missing values for any rows that had missing property values or interest rates, naturally, but it would compute correct numerical values in all other rows.

But what happens if you try to run the same code, but just on the `women` DataFrame (or `asians` or `age75andup`)?

Big Picture

The warning message you see when you attempt to run the code described above is an important one! It relates to the difference between a DataFrame and a *view* of that DataFrame. You can add columns to a DataFrame, but if you add to just a view, you'll receive a warning. We will discuss the details of this in class.

1.4.3 What if you don't remember CS230 very well?

I have several recommendations of resources you can use:

DataCamp

I will regularly be assigning you exercises from [DataCamp](#), some of which will review CS230 materials. If you remember everything from CS230, the first few weeks of these exercises should be easy and quick for you. If not, you will need to put in more time, but it will help you catch up.

Bentley faculty

I'm glad to meet with students who need help catching up on material from CS230 they may not remember. Please feel free to come to office hours!

I know that Prof. Masloff, who teaches CS230, made an extensive set of course notes available to her students. You may wish to review key portions of that document to help you stay caught up in MA346. If you did not have Prof. Masloff, you might consider [contacting her](#) and asking for her course notes anyway.

Stack Overflow

The premiere question and answer website for technical subjects is [Stack Overflow](#). You don't need to visit the site, though; if you do a good Google search for any specific Python or pandas question, one of the top hits will almost always be from Stack Overflow. Here are a few tips to using it well:

- When you do a search, put as many specific words related to your question as possible.
 - Be sure to mention Python, pandas, or whatever other libraries your question might touch upon.
 - If your question is about an error message, put the specific key words from the error message in your search.
- When viewing questions and answers on Stack Overflow, don't just accept the top answer; see if later answers might be better suited to you.

O'Reilly books

You have free access to O'Reilly Online Learning through the Bentley Library. They are one of the top publishers of high-quality tutorial books on technical subjects. To get started, [visit this page and at the bottom choose to download a mobile app for your phone or tablet](#).

Then browse their book catalog and see what looks like it might be good for you. I recommend starting here:

- Python Data Science Handbook by Jake VanderPlas, chapter 3 (or perhaps start earlier if you need to)
- Python for Data Analysis by Wes McKinney, chapter 5 (or perhaps start earlier if you need to)

Official documentation

Official documentation is used mostly for reference. It does not make a good tutorial or lesson. But it is the definitive reference, so I mention it here.

- [Python official documentation](#)
- [pandas official documentation](#)

1.4.4 Python review 2: mathematical exercises

As before, do these exercises in a new notebook in Deepnote or Colab, and when you're done, share the link to the published version into our class's Teams chat.

Exercise 2

If r is the annual interest rate and P is the principal, we're all familiar with the standard formula for the present value after n periods, $P(1 + r)^n$. Write this as a Python function. Also consider:

1. How many inputs does it take and what are their data types?
 2. What is the data type of its output?
 3. Evaluate your function on $P = 1,000$, $r = 0.01$, and $n = 7$. Ensure you get approximately \$1,072.14.
-

Exercise 3

Create a pandas DataFrame with two columns. The first column should be entitled F for Fahrenheit, and should contain the numbers from 0 to 100, counting by fives. The next column should be entitled C for Celsius, and contain the corresponding temperature in degrees Celsius for the number in the first column. Display the resulting table in the notebook.

Exercise 4

The NumPy function `np.random.randint(a, b)` picks a random integer between a and $b - 1$. Use that to create a function that behaves as follows:

- Your function takes as input a positive integer n , how many times to “roll the dice.”
- Each roll of the dice simulates two dice being rolled (each with a number from 1 to 6) and adds the results together (thus generating a number between 2 and 12).
- After all n rolls, return a pandas DataFrame with three columns:
 1. the numbers 2 through 12
 2. the number of times that number showed up
 3. the percentage of the time that number showed up
- Ensure the resulting DataFrame is sorted by its first column.

1.4.5 Functional-style code vs. imperative-style code

As you wrote the functions above, you might have found yourself falling into one of two styles. To see examples of each style, let's consider the definition of the statistical concept of *variance*. The variance of a list of data x_1, \dots, x_n is defined to be

$$\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1},$$

where we write \bar{x} to mean the mean of the data, and we pronounce it “ x bar.” If we take that function and convert it directly into Python, we might write it as follows.

```
import numpy as np

def variance_style_1 ( data ):
    return sum( [ ( x - np.mean(data) )**2 for x in data ] ) / ( len(data) - 1 )

test_data = [ 5, 10, 3, 9, -1, 5, 3, 1 ]
variance_style_1( test_data )
```

```
13.982142857142858
```

Although this function computes the variance of a list of data correctly, it piles up a lot of parentheses and brackets that some readers find unnecessarily confusing when reading code. We can make the function less compact and more explanatory by breaking the nested parentheses into several different lines of code, each storing its result in a variable. Here is an example.

```
def variance_style_2 ( data ):
    n = len(data)
    xbar = np.mean( data )
    squared_differences = [ ( x - xbar )**2 for x in data ]
    return sum( squared_differences ) / ( n - 1 )
```

(continues on next page)

(continued from previous page)

```
variance_style_2( test_data )
```

```
13.982142857142858
```

I call the first one *functional style* because we’re composing a lot of functions, each inside another. I call the second one *imperative style* because this is a programming term used to describe a line of code that gives a command; here we’ve broken the formula out into three separate commands to create variables, followed by the final formula.

Neither of these is always right or always wrong. For a short formula, you probably just want to use functional style. But for a long formula, imperative style has these advantages:

- You can use good, descriptive variable names to clarify for the reader of your code what it’s computing in each step.
- If the code you’re writing isn’t inside a function, you can split imperative-style code over multiple cells, and put explanations in between.
- If you know the reader of your code is new to coding (such as a new teammate in your organization) then imperative style gives them small pieces of code to digest one at a time, rather than a big pile of code they must understand all at once.

So consider using each style for those situations that it fits best.

1.5 Before and After

See also the slides that summarize a portion of this content.

The phrase “before and after” has two meanings for us in MA346.

- First, it relates to code: What *requirements* do we need to satisfy *before* doing something with data, and what *guarantees* do the math and stats techniques we use provide *after* we’ve used them?
- Second, it relates to communicating about code: When we’re writing explanations about our code, how do we know what kind of explanations to insert *before and after* a piece of code?

Let’s look at each of these meanings separately.

1.5.1 Requirements and Guarantees

Requirements

Almost nobody ever writes a piece of code with no clear purpose in mind. You can’t write code the way you can doodle in the margins of a notebook, aimless, purposeless, spacing out. Code almost always accomplishes something; that’s what it was built for and that’s why we use it. So when we’re coding, it’s helpful to think about our code in a purposeful way. It helps to do so in a “before and after” way.

Before writing a piece of code, you need to know what situation you’re currently in (including your data, variables, files, etc.). This is because the code you write will almost certainly have requirements that need to be true before that code can be run. Here are some examples:

- If I’m going to sort a health care DataFrame by the “heart rate” column, the DataFrame had better have a “heart rate” column, not a “heart_rate” column, or a “HeartRate” column, etc. (This is a requirement imposed by the sorting routine. It can’t guess the column name’s correct spelling; you have to provide it.)

- If I'm going to fit a linear model to the relationship between the "heart rate" variable and the "oxygen replacement" variable, I should be sure that the relationship between those two variables appears to be approximately linear. (This is a requirement imposed by the nature of linear models. It isn't always a smart idea to use a linear model if that doesn't reflect the actual relationship in the data.)

Any code I'm about to run has *requirements* that must be true in order for that code to work, and if those requirements aren't satisfied, the code will either give you an error or silently do the wrong thing. Sometimes these are called "assumptions" instead of requirements, because the code assumes you're running it in a situation where it makes sense to do so.

For instance, in the "heart rate" example above, we would get an error, because the column we tried to sort by didn't exist. But in the linear model example above, we would get no error, just a linear model that probably wasn't very useful, or might produce poor predictions.

You can think of these requirements as **what to know before running your code** (or what to check if you don't yet know it). They are almost always phrased in terms of the inputs to the function you're about to run, such as the data type the input must have, or the size/shape it must have, or the contents it must have.

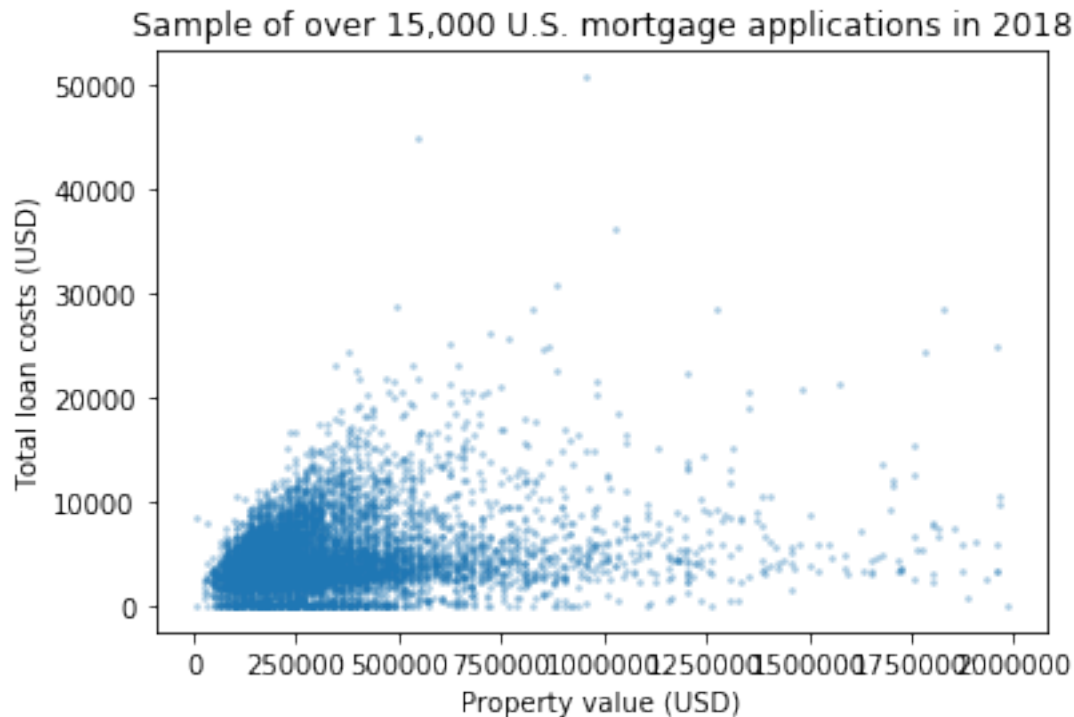
How do we avoid messing this up? *Know what the relevant requirements are* for the code you're about to run and *check them before you run the code*. In some cases, the requirements are so small that it doesn't make sense to waste time checking them, as in the "heart rate" example above. (If we get it wrong, the error message will tell us, and we'll fix it, nice and easy.) But in other cases, the requirements are important and take time to check, as in the linear model example above. In fact, let's see how that would work:

Let's say we've loaded a dataset of mortgages, with columns for `property_value` and `total_loan_costs`.

```
import pandas as pd
df = pd.read_csv( '_static/practice-project-dataset-1.csv' )
```

I'm suspecting `total_loan_costs` can be estimated pretty reliably with a linear model from `property_value`. But before I go and fit such a model, I had better check to be sure that the relationship between those variables actually seems to be linear. The code below does so.

```
import numpy as np
import matplotlib.pyplot as plt
two_cols = df[['property_value', 'total_loan_costs']].replace( 'Exempt', np.nan )
two_cols = two_cols.dropna().astype( float )
two_cols = two_cols[two_cols['property_value'] < 2000000]
plt.scatter( two_cols['property_value'], two_cols['total_loan_costs'], s=3, alpha=0.
↪25 )
plt.title( 'Sample of over 15,000 U.S. mortgage applications in 2018' )
plt.xlabel( 'Property value (USD)' )
plt.ylabel( 'Total loan costs (USD)' )
plt.show()
```



Hmm... While some portions of that picture are linear (such as the top and bottom edges, as well as a thick strip at about $y = 4000$), it's pretty clear that the whole shape is not at all close to a straight line. Any model that predicts total costs just based on property value is going to be an unreliable predictor. I almost certainly don't want to make a linear model for this after all (unless I'm in a situation in which I just need an *extremely* rough estimate). Good thing I checked the requirements before making the model!

Guarantees

Each piece of code you run also provides certain guarantees that it will do for you (as long as you took care to ensure that the assumptions it required held true). Here are some examples:

- If you have a pandas DataFrame `df` containing numeric data and you call `df.mean()`, you will get a list of the mean value of each column in the data, computed separately, using the standard definition of mean from your intro stats class.
- If you fit a linear model to data using the standard method (ordinary least squares), then you know that the resulting model is the one that minimizes the sum of the squared residuals. In other words, the expected estimation error on your data is as small as possible.

These guarantees are, in fact, the reason we run the code in the first place. We have goals for our data work, and someone has provided us some Python-based tools that help us achieve our goals. We trust the guarantees their software provides, and so we use it.

It's important to be familiar with the guarantees provided by your math and stats software, for two reasons. First, obviously, you can't choose which code to run unless you know what it's going to do when you run it! But secondly, you're going to want to be able to write good explanations to go along with your code, and you can't do that unless you can articulate the guarantees your code makes. Let's talk about good explanations next.

1.5.2 Communication

Big Picture

The best code notebooks explain their contents according to two rules:

1. Before each piece of code, explain the motivation for the code.
2. After each piece of code, explain what the output means.

Connect the two! Your output explanation should directly address your motivation for running the code.

This is so important that we should see some examples.

Example 1

Imagine that you just came across the following code, all by itself.

```
df['state_code'].value_counts().head( 10 )
```

```
CA      1684
FL      1136
TX      1119
PA       564
GA       558
OH       542
NY       535
NC       524
IL       508
MI       469
Name: state_code, dtype: int64
```

Seeing this code naturally causes us to ask questions like: Why are we running this code? What is this output saying? Who cares? What are the numbers next to the state codes? Why just these 10 states?

If instead the writer of the code had followed the two rules in the “Big Picture” lesson from earlier in the chapter, none of those questions would arise. Here’s how they could have done it:

Which states have the most mortgage applications in our dataset?

```
df['state_code'].value_counts().head( 10 )
```

```
CA      1684
FL      1136
TX      1119
PA       564
GA       558
OH       542
NY       535
NC       524
IL       508
MI       469
Name: state_code, dtype: int64
```

Each state is shown next to the number of applications from that state in our dataset, largest first, then descending. Here we show just the top 10.

Even with just a small piece of code, notice how easy it is to understand when we have the two explanations. The sentence before the code asks an easy-to-understand question that shows the writer’s motivation for the code. The two sentences after the code explain what the output shows and why we can trust it.

We help the reader out (and ourselves later when we come back to this code!) by following those two simple rules of explanation.

Example 2

Imagine encountering this code:

```
rates = df['interest_rate']
rates.describe()
```

```
count      10061
unique         500
top         4.75
freq         912
Name: interest_rate, dtype: object
```

Although in this case, you might know what’s going on because `.describe()` is so common in pandas, it still doesn’t tell us why the code was run, or what we’re supposed to pay attention to in the output.

Imagine instead that the writer of the code had done this:

We’d like to use the interest rates in the dataset to do some computation. What format are they currently stored in?

```
rates = df['interest_rate']
rates.describe()
```

```
count      10061
unique         500
top         4.75
freq         912
Name: interest_rate, dtype: object
```

The interest rates are written as percentages, since we see the most common one was 4.75 (instead of 0.0475, for example). However, they are currently stored as text (what pandas calls “dtype: object”), so we must convert them before using them. We stored them in the `rates` variable so we can manipulate it further later.

Now we know why the original coder cared about this output (and perhaps why we should). Also, if we didn’t know what “dtype: object” meant, or why we might pay attention to that, now we know. Also, we know not to multiply anything by these interest rates without also dividing by 100, because they’re percentages. Much more helpful than just the code alone!

Poor or missing explanations decrease productivity. When you work on a project that takes more than one day to do (and you will definitely have that experience in MA346), you’re guaranteed to come back and look at some code that you wrote in the past and scratch your head, wondering why it doesn’t look familiar. This happens to everyone. Help yourself out by

adding explanations about each piece of code you write. This is a requirement for the projects you do in this class; you'll see more about this when you read the specific grading requirements for each project.

If one day you find yourself coding in a professional environment, you'll definitely want to document your work with comments and explanations. You're sure to share your work with teammates at some point. You may even use your work to show new people who join the team how to get started. A pile of code without explanations is far less useful than code interspersed with careful explanations.

Knowing your target audience

When you're considering adding explanations to your code, imagine yourself explaining the code to a future reader.

- If you suspect it's a teammate that will read your code, write what you would say to them if you had to explain the code in person.
- If you know it's your MA346 instructor who will read your code, write in such a way that you prove you know what your code does and can articulate why you wrote it.
- If you know it's a new coder who will read your code, be more thorough and don't take any knowledge for granted. Think about what might confuse them and address it.

Professionalism

In a business context, taking the time required to make your writing as brief as possible has many benefits. It enhances productivity because your writing is faster to read. It reduces confusion because long writing makes people space out. It shows respect because you've invested the time required to make sure your writing doesn't waste your reader's time. Short, simple writing doesn't make you look unintelligent; it makes you look like a clear writer.

It is also essential to proofread what you've written. Code explanations that don't make sense because of typos, missing words, spelling errors, or enormous paragraphs are helpful to almost no one. Take the time to ensure your writing would make your EXP101 professor proud. In particular, any sufficiently long text (over one page, or one computer screen) needs headings to help the reader see the big picture.

Learning on Your Own - Technical Writing Tips

Interview a professor in the English and Media Studies department. Ask what their top 5 tips are for technical and/or business writing. Create a report, video, or presentation on this for your MA346 peers. Is it possible, for each tip, to show a realistic example of how bad things can be when someone disobeys the tip, compared side-by-side with a realistic example of how good things can be when the tip is followed?

Choosing a medium

Should I put my code explanations as comments in the code, or as Markdown cells, or what? Here are some brief guidelines, but there are no set rules.

- A Python script with comments in it is best if:
 - you're writing a Python module that other software developers will read (which we won't do in this class), or
 - the code is short enough that it doesn't warrant a full Jupyter notebook.
- A Jupyter notebook with Markdown cells is best if:
 - the code will generate tables and graphs that are a key part of what you're trying to communicate, and
 - the readers are other coders, who may want to see the code along with the tables and graphs,

- but it’s okay to also insert comments within code cells *in addition to* the before-and-after explanations between cells.
- A report (such as a Word doc) or slide deck is best if:
 - your audience is nontechnical and therefore will be disconcerted to see your code, or
 - your audience is technical but in this particular instance they just want your results, or
 - the amount of writing and pictures in what you need to share is high, and the amount of code very small.
 - Showing code in slides is rarely welcome in a business context.
- A code repository (which we’ll learn about in future weeks) is best if:
 - you have several files you want to share together, such as one or more notebooks and one or more data files, and
 - you know that your audience may want to have access not just to your results, but to your code and data as well, and
 - you know that your audience is comfortable accessing a code repository.

1.6 Single-Table Verbs

See also the slides that summarize a portion of this content.

The function we’ll discuss today got the name “verbs” because coders in the R community developed what they call a “grammar” for data transformation, and the function we’ll look at today are some of that grammar’s “verbs.” The origins in R are unimportant for our course; what matters is that verbs are things you can *do* with tables of data.

1.6.1 Tall and Wide Form

The following two tables show the same data, but in different forms. One is tall while the other is wide.

Tall form:

First	Last	Day	Sales
Amy	Smith	Monday	39
Amy	Smith	Tuesday	68
Amy	Smith	Wednesday	10
Bob	Jones	Monday	93
Bob	Jones	Tuesday	85
Bob	Jones	Wednesday	0

Wide form:

First	Last	Monday	Tuesday	Wednesday
Amy	Smith	39	68	10
Bob	Jones	93	85	0

Although it’s not part of MA346, it’s worth mentioning that: In the famous paper [Tidy Data](#), data scientist and R developer Hadley Wickham calls tall form “tidy data” and defines it as having exactly one “observation” per row. (What an observation is depends on what you’ve gathered data about. In the first table above, an observation seems to be the amount of sales by a particular person on a particular day.) His rationale comes from people who’ve studied databases, and if

you've taken CS350 at Bentley, you may be familiar with the related concept of database normal forms. The [tidyverse](#) is a collection of R packages that help you work smoothly with data if you organize it in tidy form.

Big Picture

The tall form is typically more useful when computing with data, because we often want to filter for just the rows we care about. So the more separated the data is into rows, the easier it is to select just the data we need.

The wide form is typically more useful when presenting data to humans. Although this tiny table is just an example, data in the real world has far more rows, meaning that the tall form will not fit on a page. Reshaping it into a rectangle that does fit on one page is usually preferred.

Pivot is the verb that converts tall form to wide form.

Melt is the verb that converts wide form to tall form.

Let's investigate them.

1.6.2 Pivot

As just stated, pivot is the verb for converting tall-form data to wide-form data. We'll give a precise definition later on. Let's first get some intuition for how it works.

The general idea

The big picture idea of the pivot operation is illustrated here:

We will make that more precise later, but it can serve as a reference for the general idea.

The table below shows the same table from above, in "tall" form. Drag the slider back and forth to watch the transition from tall to wide form. While you do so, watch each of these parts of the table:

1. The gray cells:
 - These are the unique IDs used in both shapes, tall or wide.
 - They function like row headers.
 - In pandas, we call them the `index` of the pivot.
2. The blue cells:
 - The most important change happens here.
 - In tall form they're data, but in wide form they're column headers.
 - In pandas, we call them the `columns` of the pivot (because they turn into columns when we pivot).
3. The green cells:
 - These contain the values, typically numbers.
 - They do not change, but merely move to sit in the appropriate place in each table.
 - In pandas, we call them the `values` of the pivot.

(This animation can be viewed [in its own page here](#).)

```
from IPython.display import IFrame
IFrame( 'https://nathancarter.github.io/dataframe-animations/pivot.html?hide-
↪title=true',
        width=700, height=800 )
```

```
<IPython.lib.display.IFrame at 0x11b48c6a0>
```

The precise definition

We can state precisely what `df.pivot()` does by building on what we've learned in previous chapters. We can describe both the *requirements* and *guarantees* of the pivot function, and can do so in terms of functions and relations.

- Requirements of `df.pivot()`:
 - The table to be pivoted must express a *function* from at least two input columns (called `index` and `columns`, above) to one output column (called `values`, above).
 - It is acceptable for the `index` to comprise more than one column, as in the example above.
 - Recall that for it to be a function, inputs cannot be repeated, because that could connect them with more than one output.
- Guarantees of `df.pivot()`:
 - Each value from the `index` columns will appear only once in the resulting table.
 - A new column will be created for each unique value in the old `columns` column.
 - The `values` column will have been removed.
 - For each `index` entry i in the original DataFrame and each `columns` entry c , if v is the unique value associated with it, then the new table will contain a row with `index` i and with v in the column entitled c .

You can think of `df.pivot()` as turning one function into many. In the example above, it worked like this:

- Original table
 - One function
 - * Inputs: first name, last name, day
 - * Output: sales
- Result of pivoting
 - First function
 - * Inputs: first name, last name
 - * Output: Monday sales
 - Second function
 - * Inputs: first name, last name
 - * Output: Tuesday sales
 - Third function
 - * Inputs: first name, last name
 - * Output: Wednesday sales

Purpose of pivoting

Recall that pivoting just turns “tall” data into “wide” data. And tall form is how you typically store data when doing an analysis, because of the ease of processing tall data using code, while wide form is often more attractive for a human reading data from a table. **So the purpose of pivoting is typically when you’re generating reports for human consumption.**

1.6.3 Melt

The reverse operation to a pivot is called “melt.” This comes from the fact that wide data “falls down” (like the drips of a melting icicle perhaps?) into tall form. The idea is summarized in the following picture, but you can watch it happen in the animation further below.

The genreal idea

The big picture idea of the pivot operation is illustrated here:

We will make that more precise later, but it can serve as a reference for the general idea.

Just as pivoting was usually to turn data stored for computers into data readable by humans, melting is for the reverse. If you’re given data in wide form, but you want to prepare it for analysis, you often want to convert it into tall form to make subsequent data processing code easier.

For example, let’s say we were given the table below of students’ performance on various exams. (Obviously, this is fake data.) If we would rather view each exam as a separate observation, so that each row is a single exam score, we can melt the table.

Drag the slider to see the melting in action. While you do so, watch the following parts of the table:

1. The gray cells:
 - Because we’ll be spreading a student’s data out over more than one row, these will be copied.
 - These function as unique IDs for each row, so pandas calls these columns the `id_vars`.
2. The blue cells:
 - These are the titles for each of several different functions.
 - Each function takes a student as input and gives a type of exam score as output.
 - They will change from being column headers to being values in the table, so pandas calls them the `value_vars`.
3. The green cells:
 - Each column represents a separate function (the first maps students to SAT score, the second maps students to ACT score, and the third maps students to GPA).
 - Because we’re collecting all scores into a single column, these will stack up to become just one column.

(This animation can be viewed [in its own page here](https://nathancarter.github.io/dataframe-animations/melt.html?hide-title=true).)

```
from IPython.display import IFrame
IFrame( 'https://nathancarter.github.io/dataframe-animations/melt.html?hide-title=true' ,
        width=700, height=900 )
```

```
<IPython.lib.display.IFrame at 0x11b376358>
```

The precise definition

Unsurprisingly, the requirements and guarantees of the melt operation are the reverse of those from the pivot operation.

- Requirements of `df.melt()`
 - The `id_vars` are one or more columns that contain unique identifiers for each row.
 - The `value_vars` columns are each a function from the `id_vars`. (That is, no value in `id_vars` appears twice.)
- Guarantees of `df.melt()`
 - For each value i in the `id_vars` column and for each column c in the `value_vars`, if we write f for the function that column represents, then the new table will contain a row with ID i and values c and $f(c)$.
 - This new table will therefore be a function from the i and c columns to the $f(c)$ column. (By default, pandas calls those two new columns “variable” and “value” but you can give them more meaningful names.)
 - There are no other rows in the resulting table besides those just described.

1.6.4 Pivot tables

All this talk of pivoting should remind you of the very common Excel operation called “pivot table.” It is very much like the pivot operation, with two differences. First, it doesn’t require the table to represent a function. Second, it does require you to explain how values will be summarized or combined. Naturally, pandas supports this operation as well, and it’s extremely useful.

If `df.pivot()` makes a tall table wide, then `df.pivot_table()` makes a tall table sort of wide. We’ll see why below.

The general idea

The big picture idea of the pivot operation is illustrated here:

We will make that more precise later, but it can serve as a reference for the general idea.

In the table shown below, notice that if we try to consider the gray and blue columns as inputs and the green column as outputs, the relationship is *not* a function. If it were, we could pivot on the blue column, and the green cells would rearrange themselves just as they did in the first animation up above. But try dragging the slider below *slowly* and you will see that some green cells collide.

For instance, Amy Smith has two different sales to the same customer, Facebook, and Bob Jones has two different sales to the same customer, Amazon. So we cannot simply create a Facebook column and an Amazon column and rearrange the sales data into them. When two sales figures need to be placed under the same customer heading, we need some way to combine them.

The way the table below combines cells is by adding, which is a very sensible thing to do with sales data for a customer. You can see that the code asks this by specifying the aggregation function (or `aggfunc`) to be “sum.”

This is why a `pivot_table` operation doesn’t make a table that’s as wide as a `pivot` might, because some cells are combined, meaning that the overall table reduces in size.

(This animation can be viewed [in its own page here](#).)


```
from IPython.display import IFrame
IFrame( 'https://nathancarter.github.io/dataframe-animations/pivot-table.html?hide-
↩title=true',
        width=800, height=800 )
```

```
<IPython.lib.display.IFrame at 0x118ede2e8>
```

The precise definition

I will alter the precise definition of `df.pivot()` as little as possible when creating this definition of `df.pivot_table()`.

- Requirements of `df.pivot_table()`:
 - The table to be pivoted can express any *relation* among least three columns (called `index`, `columns`, and `values`, above).
 - It is acceptable for the `index` to comprise more than one column, as in the example above. (Same as for `df.pivot()`.)
 - We must have some aggregation function (called `aggfunc`, above) that can combine many entries from the `values` column into one. In the example above, we used “sum.” Let’s call this function A .
- Guarantees of `df.pivot_table()`:
 - Each value from the `index` columns will appear only once in the resulting table. (Same as for `df.pivot()`.)
 - A new column will be created for each unique value in the old `columns` column. (Same as for `df.pivot()`.)
 - The `values` column will have been removed. (Same as for `df.pivot()`.)
 - For each `index` entry i in the original DataFrame and each `columns` entry c , if v_1, v_2, \dots, v_n are the various values associated with it, then the new table will contain a row with `index` i and with $A(v_1, v_2, \dots, v_n)$ in the column entitled c .

1.6.5 Stack and unstack

There are two other single-table verbs that you studied in the DataCamp review before today’s reading. These are less common because they apply only in the context where there is a multi-index, either on rows or columns. But we give animations of each below to help the reader visualize them.

The `stack` operation takes nested column indices (which are arranged horizontally) and makes them nested row indices (which are arranged vertically). This is why it’s called “stack,” because it arranges the headings vertically. `unstack` is the same operation in reverse.

When applying these operations, it is possible to choose which level of a multi-index gets stacked or unstacked. The two animations below use two different levels, so that you can compare the differences.

Animation for unstack/stack at level 1

The level of a stack/unstack operation refers to which level of the multi-index will be moved. The animation below shows `df.unstack(level=1)` when you move the slider from left to right, so level 1 of the row multi-index (the weeks) moves up to become part of the column index. It is always placed as an inner index, but this can be changed afterwards with `df.swaplevel()`.

The reverse operation is exactly `df.stack(level=1)`, because it moves level 1 from the column headings back to be inside the row headings instead.

(This animation can be viewed [in its own page here.](#))

```
from IPython.display import IFrame
IFrame( 'https://nathancarter.github.io/dataframe-animations/stack-1.html?hide-
↪title=true',
        width=800, height=900 )
```

```
<IPython.lib.display.IFrame at 0x11b49b9e8>
```

Animation for unstack/stack at level 0

The level of a stack/unstack operation refers to which level of the multi-index will be moved. The animation below shows `df.unstack(level=0)` when you move the slider from left to right, so level 0 of the row multi-index (the months) moves up to become part of the column index. It is always placed as an inner index, but this can be changed afterwards with `df.swaplevel()`.

The reverse operation is therefore actually a combination of `df.stack()` (which would put the months inside the weeks) and `df.swaplevel()` (which would fix that) all in one.

(This animation can be viewed [in its own page here.](#))

```
from IPython.display import IFrame
IFrame( 'https://nathancarter.github.io/dataframe-animations/stack-0.html?hide-
↪title=true',
        width=700, height=900 )
```

```
<IPython.lib.display.IFrame at 0x107f714e0>
```

1.7 Abstraction

See also the slides that summarize a portion of this content.

1.7.1 Abstract vs. concrete

Abstract/concrete are opposite ends of a spectrum:

	Concrete (or specific)	Abstract (or general)
Example from science:	When we drop things, they fall to earth.	$G_{\mu\nu} + \Lambda g_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu}$ (Einstein's field equation)
Example from business:	That startup failed because each partner tried to pull it in a different direction.	Organizations need a clearly stated vision.
Example from ethics:	The Nazis' attacks on the Jews were a great evil.	Systematically disadvantaging any racial group is wrong.

Abstraction is therefore the process of moving from the concrete toward the abstract, or from the specific to the general. Therefore it's also called *generalization*. Humans are pretty good at learning general principles from specific examples, so this is a natural thing for us to do.

It's very useful in all kinds of programming, including data-related work, so it's our focus in this chapter.

1.7.2 Abstraction in mathematics

Example 1: Algebra class

My kids are teenagers and have recently taken algebra classes where they learned to “complete the square.” This procedure takes a quadratic equation like $16x^2 - 9x + 5 = 0$ and manipulates it into a form that's easy to solve.

- Each homework problem was a *specific* example of this technique.
- If you apply the technique to the equation $ax^2 + bx + c = 0$, the result is the quadratic formula, $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$, a *general* solution to all quadratic equations.

Abstraction from the specific to the general tends to create more powerful tools, because they can be applied to any specific instance of the problem.

Example 2: Excel formulas

If you took the grading policy out of the syllabus for this class, you could compute your grade in the course based on your scores on each assignment. You could do this by hand with pencil and paper, or with a calculator. Doing so would give you one specific course grade, for the specific assignment grades you started with.

Alternately, you could fire up a spreadsheet like Excel, and create cells for each assignment's score, then create formulas that would do the appropriate computation and give you the corresponding course grade. This general solution works for any specific assignment grades you might type into the spreadsheet's input cells.

Again, the general version is more useful.

Observations

Both of these mathematical examples involved replacing numbers with variables. In Example 1, the coefficients in the specific example $16x^2 - 9x + 5 = 0$ turned into a , b , and c in $ax^2 + bx + c = 0$. In Example 2, you didn't write formulas that had specific scores in them (as you would have if computing the scores by hand), but wrote formulas that contained Excel-style variables, which have names like A5 and B14, that come from the relevant cells. In math (and in programming as well), abstraction typically involves *replacing specific constants with variables*.

Once we've rephrased our computation in terms of variables, we can do many different mathematical operations with it.

1. We can think of our computation as a function.

- In Example 1, the quadratic formula can be seen as a function that takes as input the values a , b , c and yields as output the two solutions of the equation.

- In Example 2, the Excel formulas can be seen as a function that take the assignment grades as input and yield the course grade as output.
2. We can ask what happens when one of the variables changes, a question that calculus focuses on.
 - For instance, you could ask what happens to your computation as one of the variables gets larger and larger. (In calculus, we wrote this as $\lim_{x \rightarrow \infty}$.)
 - Or you could ask how the result of the computation responds to changes in one input variable. (In calculus, we wrote this as $\frac{d}{dx}$.)
 3. We can make statements about the computation in terms of the input variables.
 - In Example 1, we might say that “Every quadratic equation has two complex number solutions.”
 - In Example 2, we might say that “It’s still possible for me to get a 4.0 in this course if my final exam score is good enough.”

The statement above from Example 1 is a *universal* statement, also called a “for all” statement. You could rephrase it as: For all inputs a, b, c , the outputs of the quadratic formula are two complex numbers. The statement from Example 2 is an *existence* statement, also called a “for some” statement. You could rephrase it as: For some final exam scores, my final course grade is still a 4.0. For all/for some statements are central to mathematics and we will see them show up a lot. “For all” and “for some” are called *quantifiers* and are sometimes written \forall (for all) and \exists (for some, or “there exists”).

1.7.3 Abstraction in programming

Big Picture

This section covers the value of abstraction for every programmer. It is a valuable viewpoint to have and skill to be able to employ. See the rest of this section for details on how it works and how to use it.

Example 3: Copying and pasting code

Best practices for coding include writing DRY code, where DRY stands for Don’t Repeat Yourself. If you find yourself writing the same code (or extremely similar code) more than once, especially if you’re copying and pasting, this is a sure sign that you are not writing DRY code and should try to correct this style error. The way to correct it is with abstraction, as shown below. (The opposite of DRY code is WET code—Write Everything Twice. Don’t do that.)

Here is an example of some code a student once wrote for me in a past data science course. They had three pandas DataFrames of data about COVID-19, one containing numbers of cases, one containing numbers of deaths, and one containing numbers of recoveries. They wanted to change the column names in each DataFrame.

```
df_cases = df_cases.add_suffix( '_cases' )
df_cases.columns = ['Province/State', 'Country/Region', 'Lat', 'Long'] + list(df_cases.
↪columns[4:])
df_deaths = df_deaths.add_suffix( '_deaths' )
df_deaths.columns = ['Province/State', 'Country/Region', 'Lat', 'Long'] + list(df_deaths.
↪columns[4:])
df_recoveries = df_recoveries.add_suffix( '_recoveries' )
df_recoveries.columns = ['Province/State', 'Country/Region', 'Lat', 'Long'] + list(df_
↪recoveries.columns[4:])
```

I suspect the student wrote the first two lines, ran them, verified that they worked, and then copied and pasted them twice and changed the variable names to apply the same code to the other two DataFrames. But this code can be made much cleaner through abstraction. Rather than copy and paste the code, then change key parts of it, replace those key parts with

a *general* (that is, abstract) variable name, and then turn the code into a function. Since this code renames columns, the student could have made that the name of the function.

```
def rename_columns ( df ):
    df = df.add_suffix( '_cases' )
    df.columns = [ 'Province/State', 'Country/Region', 'Lat', 'Long' ] + list(df.
↳ columns[4:])
```

Once this general version is complete, you can apply it to each specific case you need.

```
rename_columns( df_cases )
rename_columns( df_deaths )
rename_columns( df_recoveries )
```

There are several advantages to this new version.

1. While it is still six lines of code, half of them are much shorter, so there's less to read and understand.
2. What the code is doing is more obvious, because we've given it a name; we're obviously renaming columns.
3. It wasn't immediately obvious in the first version of the code that we were repeating the same procedure three times. Now it is.
4. If you later need to change how you rename columns, you have to make that change in only one place (inside the function). Before, you would have had to make the same change three times.
5. Also, if you tried to make a change to the code later, but accidentally missed changing one of the three, you'd have broken code and not realize it.
6. You could share this same function to other notebooks or with other coders if needed.

So the moment you find yourself copying and pasting code, remember to stay DRY instead—create a function and call it multiple times, so that you get all these benefits.

Alternatives

Another method of abstraction would have been a loop instead of a function. Since the original code does the same thing three times, we could have rewritten it as follows instead.

```
for df in [ df_cases, df_deaths, df_recoveries ]:
    df = df.add_suffix( '_cases' )
    df.columns = [ 'Province/State', 'Country/Region', 'Lat', 'Long' ] + list(df.
↳ columns[4:])
```

This has all the same benefits as the previous method, except for #6.

One could even combine the two methods together, as follows.

```
def rename_columns ( df ):
    df = df.add_suffix( '_cases' )
    df.columns = [ 'Province/State', 'Country/Region', 'Lat', 'Long' ] + list(df.
↳ columns[4:])

for df in [ df_cases, df_deaths, df_recoveries ]:
    rename_columns( df )
```

Example 4: Testing a computation

Let's imagine that the same student as above, who has COVID-19 data, wants to investigate its connection to the polarized political climate in the U.S., since COVID-19 response has become very politicized. They want to ask whether there's any correlation between the spread of the virus in a state and that state's prevailing political leaning. So the student gets another dataset, this one listing the percentage of registered Republicans and Democrats in each U.S. state. They will want to look up each state in the COVID-19 dataset in this new dataset, to connect them. They try this:

```
# load political data
import pandas as pd
df_pol_reg = pd.read_excel( '_static/political-registrations.xlsx',
                           sheet_name=0 )

# make dictionaries for easy lookup:
state_rep_pct = dict( zip( df_pol_reg['State'], df_pol_reg['R%'] ) )
state_dem_pct = dict( zip( df_pol_reg['State'], df_pol_reg['D%'] ) )

# see if it works on Alaska:
state_rep_pct['AK']
```

0.26

Great, progress! Let's just try one or two more random examples to be sure that wasn't a fluke.

```
# see if it works on Alabama:
state_rep_pct['AL']
```

```
-----
KeyError                                Traceback (most recent call last)
<ipython-input-2-012bacaf2696> in <module>
      1 # see if it works on Alabama:
----> 2 state_rep_pct['AL']

KeyError: 'AL'
```

Uh-oh. Checking the website where they got the data, the student finds that Alabama doesn't register voters by party, so Alabama isn't in the data. They need some code that won't cause errors for any state input, so they update it:

```
import numpy as np
state_rep_pct['AL'] if 'AL' in state_rep_pct else np.nan
```

nan

Great, this looks like it will work for any input. Let's turn it into a function and test that function.

```
def get_rep_pct ( state_code ):
    return state_rep_pct[state_code] if state_code in state_rep_pct else np.nan
def get_dem_pct ( state_code ):
    return state_dem_pct[state_code] if state_code in state_dem_pct else np.nan
get_rep_pct( 'AK' ), get_rep_pct( 'AL' ), get_dem_pct( 'AK' ), get_dem_pct( 'AL' )
```

(0.26, nan, 0.14, nan)

So Example 4 has shown us that abstracting a computation into a function can be done as part of an ordinary coding workflow: Start easy, by doing the computation on just one input and get that working. Once it does, test it on some other inputs. Then create a function that works in general.

The benefits of this include all the benefits discussed after Example 3, plus this one: The student wanted to run this computation for every row in a DataFrame. That's easy to do now, with code like the following.

```
df_cases['state_rep_pct'] = df_cases['Province/State'].apply( get_rep_pct )
df_cases['state_dem_pct'] = df_cases['Province/State'].apply( get_dem_pct )
```

Little abstractions (lambda)

When it would be handy to create a function, but the function is so small that it seems like giving it a name with `def` is overkill, you can use Python's `lambda` syntax to create the function.

(The name comes from the fact that some branches of computer science use notation like $\lambda x.3x+1$ to mean “the function that takes x as input and gives $3x+1$ as output.” So they could write $f = \lambda x.3x+1$ instead of $f(x) = 3x+1$.)

For example, let's say you have a dataset in which each row represents an hour of trading on an exchange, and the volume is classified using the codes 0, 1, 2, and 3, which stand (respectively) for low volume, medium volume, high volume, and unknown (missing data). We'd like the dataset to be more readable, so we'd like to replace those numbers with the actual words low, medium, high, and unknown. We could do it as follows.

```
def explain_code ( code ):
    words = [ 'low', 'medium', 'high', 'unknown' ]
    return words[code]

df['volume'] = df['volume'].apply( explain_code )
```

But this requires several lines of code to do this simple task. We could compress it into a one-liner as follows.

```
df['volume'] = df['volume'].apply( lambda code: ['low', 'medium', 'high', 'unknown'
↪][code] )
```

The limitation to Python's `lambda` syntax is that you can put inside only a single expression, which the function will return. A function that needs to do several preparatory computations before returning an answer cannot be converted into `lambda` form.

1.7.4 How to do abstraction

If you aren't sure how to take specific code and turn it into a general function, I suggest following the steps given here. Once you've done this a few times, it will come naturally, without thinking through the steps.

Let's use with the following example code to illustrate the steps. It's useful in DataFrames imported from a file where dollar amounts were written in a form like \$4,320,000.00, which pandas won't recognize as a number, because of the commas and the dollar sign. This code converts such a column to numeric. Since it's so useful, we may want to use it on multiple columns.

```
df['Tuition'] = df['Tuition'].str.replace( "$", "" ) # remove dollar signs
df['Tuition'] = df['Tuition'].str.replace( ",", "" ) # remove commas
df['Tuition'] = df['Tuition'].astype( float )       # convert to float type
```

Step 1: Decide which parts of the code are customizable. That is, which parts of the code might change the next time you want to use it?

In this code, we certainly want to be able to specify a different column, so 'Tuition' needs to be customizable. Also, we've converted this column to type `float`, but perhaps some other column of money might better be represented as `int`, so we'll let the type be customizable also.

Step 2: Move each of the customizable pieces of code out into a variable with a helpful name, declared before the code is run.

This is probably clearest if it's illustrated:

```
column = 'Tuition'
new_type = float
df[column] = df[column].str.replace( "$", "" ) # remove dollar signs
df[column] = df[column].str.replace( ",", "" ) # remove commas
df[column] = df[column].astype( new_type )
```

You can then re-run this code to be sure it still does what it's supposed to do.

Step 3: Decide on a succinct description for what your code does, to use as the name of a new function.

In this case, we're converting a column of currency to a new type, but I don't want to call it "convert currency" because that sound like we're using exchange rates between two currencies. Let's call it "simplify currency."

Step 4: Indent your original code and introduce a `def` line to define a new function with your chosen name. Its inputs should be the names of the variables you created.

In our example:

```
def simplify_currency ( column, new_type ):
    df[column] = df[column].str.replace( "$", "" ) # remove dollar signs
    df[column] = df[column].str.replace( ",", "" ) # remove commas
    df[column] = df[column].astype( new_type )
```

If you run it at this point, it doesn't actually do anything to your DataFrame, because this just defines a function. So we need one more step.

Step 5: Call your new function to accomplish what your original code used to accomplish.

```
def simplify_currency ( column, new_type ):
    df[column] = df[column].str.replace( "$", "" ) # remove dollar signs
    df[column] = df[column].str.replace( ",", "" ) # remove commas
    df[column] = df[column].astype( new_type )

simplify_currency( 'Tuition', float )
```

This should have the same effect as the original code. Except now you can re-use it on as many inputs as you like.


```
simplify_currency( 'Fees', float )
simplify_currency( 'Books', float )
simplify_currency( 'Room and board', float )
```

Sorry, that's a depressing example. Let's move on...

1.7.5 How do I know when to use abstraction?

Whenever you find yourself copying and pasting code with minor changes, this is a sure sign that you should write a function instead. The reasons why are essentially the six benefits listed at the end of *Example 3*, above.

Also, if you have several lines of code in a row with only one thing changing, you can use abstraction to create a loop instead of a function. We saw an example of this in the *Alternatives* section, above. This is especially important if there's a numeric progression involved.

In class, we will practice using abstract to improve code written in a redundant style.

Later, the skill of abstracting code will be a crucial part of our work on creating interactive dashboards.

Learning on Your Own - Code refactoring

Some IDEs can help automate the process of abstraction. This is part of a larger set of features that such apps often call “refactoring” or “code refactoring.” Consider researching features in VS Code, PyCharm, or Eclipse that support code refactoring in Python and creating a report or video showing the class how to use those features to accomplish the content of this chapter.

Learning on Your Own - Writing Python modules

Once you've created a useful function, such as the `simplify_currency` function above, that you might want to reuse in many Python scripts or Jupyter notebooks, where should you store it? Copying and pasting it across many notebooks creates the same problems that copying and pasting any code causes. The best strategy is to create a Python module.

A tutorial on writing Python modules could answer the following questions.

- How do I start creating a Python module?
- How do I move a function I've written into my new Python module?
- Where do I store a Python module I've created?
- How do I import my new module into scripts or notebooks I write?
- How do I use the functions in my module after I've imported it?
- Can I publish my module online in an official way?

1.8 Version Control

See also the slides that summarize a portion of this content.

1.8.1 What is version control and why should I care?

Big Picture

The most common version control system is called `git`. It helps you with:

- keeping old snapshots of your work in case you need to undo a mistake
 - collaborating with others on your team by sharing a project
 - publishing your project online, for sharing or as a backup
-

It's called "version control" software because of the first of those bullet points. The other two are also important, but aren't the main purpose of `git`. Let's dive a little deeper into each of those three points and learn some terminology.

1.8.2 Details and terminology

Repositories

When you start a new project, you should make a folder to contain just the stuff for that project. By default, a folder on your computer is *not* tracked by `git`. If you want `git` to start tracking a folder and keeping snapshots, to enable the features listed above, you have to turn the folder into what is called a **git repository**, or for short, a **repo**. (You might also hear "source code repository" or "source repo" or similar terms.)

Once you do so, `git` is ready to track the changes in that folder. But it needs some direction from you. Let's see why.

Tracking changes

As you work on the project, inevitably you have ups and downs. Maybe it goes like this:

1. You start by downloading a dataset from the instructor and starting a new blank Python script or Jupyter notebook in your repo folder. Everything's fine so far. ☹️
2. You try to load the dataset but keep getting errors. You don't manage to solve it before you have to go to dinner. ☹️
3. A friend at dinner reminded you about setting the text encoding, and that fixed the problem. You get the dataset loading before bed. Yes! ☹️
4. The next day before MA346 you get the data cleaned without a problem. ☹️
5. During class, the instructor asks your team to make progress on a hypothesis test, but you run out of time in class before you can figure out all the details. The last few lines of code still give errors. ☹️

And so on. You could make the story up yourself.

If you were keeping snapshots of your work for the project, you typically wouldn't want to have any broken ones. That is, you might want to have stored your work in steps 1, 3, and 4, because if you ever had to undo some mistake you made later, you'd want to go back to a situation where you know everything was working fine. ☹️ Nobody wants to rewind to a broken repository; that's not helpful. ☹️

So you wouldn't want your version control system to automatically make snapshots for you; it would probably save a snapshot after 1, 2, 3, 4, and 5, some broken and some not. Therefore `git` doesn't do this. If you want to save a snapshot, you have to tell `git` to do so; this is called **committing** your changes. (Or sometimes you'll hear people call it **making a commit**.) When you do so, you attach a brief note (one phrase or half a sentence) describing it, called a **commit message**.

If you did so after each of steps 1, 3, and 4, above, you might have a list of commit messages that look like this:

- Downloaded dataset and started new Python script
- Wrote code to load data
- Added code to clean data

Later, if you wanted to go back to some old snapshot, `git` can show you this list of commit messages so you know exactly which one you'd like to rewind to. (At this point, I'll stop calling them "snapshots" and start using the official term, "commits.")

In fact, these course notes are stored in a `git` repository, and you can [see its list of commits online, here](#).

Sharing online

When you want to back your work up on another computer (in case yours gets broken, or if you want to publish it for others to see) there are websites that specialize in `git`. The most popular is [GitHub](#), acquired by Microsoft in 2018. In these notes, we'll teach you how to use GitHub and assume that's where you're publishing your work.

The `git` term for a site on which you back up or publish a repository is called a **remote**. This is in contrast to the repo folder on your computer, which is called your **local** copy.

There are three important terms to know regarding dealing with remotes in `git`; I'll phrase each of them in terms of using GitHub, but the same terms apply to any remote:

- For repositories you created:
 - Sending my most recent commits to GitHub is called **pushing** my changes (that is, my commits).
- For repositories someone else created:
 - Getting a copy of a repository is called **cloning** the repository. It's not the same as downloading. A download contains just the latest version; a clone contains all past snapshots, too.
 - If the original author updates the repository with new content and I want to update my clone, that's called **pulling** the changes (opposite of push, obviously).

Although technically it's possible to pull and push to the same repository, we'll come to that later. Let's start simple.

So how do we do all the things just described? The next section gives the specifics.

1.8.3 How to use `git` and GitHub

Warning: When you're reading this chapter to prepare for Week 4's class, you do not need to follow all these instructions. We will do them together in class. Feel free to just skim this section for now, and begin reading again in *the next section*.

Get a GitHub account

Do so on [this page](#) of the GitHub website. Easy!

Warning: Please choose a GitHub username that lets me know who you are. Grading will be a confusing challenge if everyone has names like `DarkKitten75XD`.

Just be sure to remember the username and password, because you'll need them in the next step.

Download the GitHub app

If you ever hear horror stories of people dealing with `git`, there are two main reasons for this. First, they may have had a repo get screwed up because multiple people were trying to edit it in conflicting ways. We will avoid such problems by focusing first on using `git` by yourself before we consider how to use it on a team project. Second, they may have been using `git`'s command-line interface, meaning they interact with it through typing commands, rather than using an app. We will avoid this hassle by getting the GitHub app.

Download and install the [GitHub app](#) from here.

When you set the app up, it will ask for the username and password of your GitHub account, so it can connect to the GitHub site.

Create a repository

Let's create a repository for you to use when submitting Project 1 later in a little over two weeks.

- If you haven't already done so, create a folder on your computer for storing your work on Project 1.
 - You don't have to put anything in the repository at all—the folder can stay empty for now.
- Using the GitHub app, turn that folder into a repository.
 - From the File menu, choose “Add local repository...” and pick the folder you just created.

Publish the repository

It's okay that your repository is still empty; you can add files later.

- In the center of the GitHub app window there should be a button called “Publish repository.”
- If not, go to the Repository menu and choose “Push.”

Warning: Ensure that you check the box to **keep the code private**. This is so that when you actually begin work on Project 1, you are not tempting anyone else to violate Bentley's academic integrity policy by looking at your work.

View it online

From the GitHub app, click the Repository menu, and “View on GitHub.” Easy! You've successfully found where the repository lives online.

Because you marked the repository private, anyone other than you who visits that page won't be allowed to see the repository. You can see it only because you've already logged in to GitHub with your username and password.

Later you'll share this repository with your instructor so that he can visit it to grade your Project 1, once it's complete.

Make a commit

In order to commit some changes to our new Project 1 repo, we have to actually do something in that folder, so there *are* some changes to commit. Let's do some simple setup.

- The Project 1 assignment on Blackboard lists three datasets you should download for use in the project. If you haven't already downloaded them, do so now. Once you've downloaded them, move them into the folder for your new repo.
- Return to the GitHub app and you should notice the three new files listed in the left column, showing you what's new in the repo since it was created.
- On the bottom left of the page, type an appropriate commit message, such as "Adding data files," and click "Commit to master."
 - You can have multiple different flavors of a project all in one repo. They're called **branches** and the main one is called the **master branch** by default.
 - In an effort to remove any potential reference to slavery, however indirect, [GitHub is in the process of changing the term "master" to "main,"](#) but that process is not yet complete as of this writing.
 - In the meantime, think of the term "master" as just a reference to the primary copy of your work.
 - You probably won't need to create any other branches in any repo in MA346.

You should see your changes disappear from the left column. This doesn't mean that they've been removed! It just means that the snapshot has been saved, so those changes aren't "new" any more. They've been committed (saved) to the repo's history.

Publish your commit

Push your changes to the repo with the Push button in the center of the app, then reload the webpage that views the repo online. You should see your new data files in the web interface. That's how easy it is to publish your work to GitHub!

Repeat as needed

Whenever you make changes to your work and want to save a snapshot, feel free to repeat the "commit" instructions you see above. The best practice is to do this as often as possible, but to try to never commit a project that's got errors or broken code. So try to make small, successful changes and commit after each one.

Warning: The GitHub app and `git` in general can see *only changes that you have saved to disk!* So if you've edited a Python script but *have not saved*, then `git`/GitHub will not be able to see those changes. The GitHub app looks only at the files on your hard drive. It does not spy on what you're doing in Jupyter or VS Code or any other app you have open.

The takeaway: Be sure to save your files to disk before you try to commit.

Whenever you want to publish your most recent commits to the GitHub site, repeat the "publish" instructions you see above.

1.8.4 What if I want to collaborate?

Collaborating with `git` is a very specific type of collaboration.

On the one hand, it's much less snappy and convenient than Google-docs-style collaboration, which happens instantaneously. You can see one another's cursors moving about the document and making edits in real time, live. (You can do this on Deepnote and CoCalc, too, in Jupyter notebooks.)

On the other hand, that's actually a good thing. If you and someone else are editing code at the same time, one of you might make changes to a variable name at the top of the file that breaks code you're writing using that variable at the bottom of the file. With `git`, you have to take intentional steps to combine two people's work, and this helps you make sure that the changes are consistent and don't lead to broken code.

Here's how you do it.

How to let someone view your private repository

You will want to do this with your Project 1 repository in two different ways.

- Recall that you're permitted to have a collaborator on Project 1 in MA346 if you want one. If so, you would add them as a collaborator using the steps below.
- Every team will share their Project 1 repository with the instructor, so that I can grade it later.

The steps for sharing a private repository with selected individuals are very straightforward:

- Visit your repository on GitHub.
- Click Settings (rightmost tab near the top of the page), then Manage access (near the top left), and Invite teams or people (bottom center).
- You'll need the GitHub username of your intended collaborator. My username on GitHub is (unsurprisingly) `nathancarter`.

To share a public repository, you can just email the link. Also, people doing a web search or viewing your GitHub profile can see all your public repositories (but not your private ones, of course).

How to have two contributors in a repository

Let's say Teammate A creates the repository and shares it with Teammate B, using the procedure described above. Then Teammate B needs to get their own local copy, like so:

- Visit the repository on the GitHub website.
- Click the green Code button, and on the menu that appears, choose Open with GitHub Desktop.
- This will launch the GitHub app and ask Teammate B to choose where on their computer they'd like to store a clone of the repository.
 - When you choose a folder, the repository will be placed as a new *folder* inside the one you choose.
 - For example, if you pick `My Documents\MA346\`, then the repository will be cloned into `My Documents\MA346\the-repo-name\`, with all the files inside that inner folder.

Then Teammate A can go off and do some work on the project and *Teammate B can do work at the same time*. They should coordinate, however, so that they don't do conflicting work. We'll come back to this in detail later.

Let's say Teammate A accomplishes some stuff and wants to commit it and share it with Teammate B. They can do this:

- Do a commit just as they ordinarily would. (See instructions up above.)
- Push that commit to GitHub just as before. (See instructions up above.)

- Tell Teammate B they have pushed, so that Teammate B knows there's new work they'll want to get.

Then Teammate B uses the GitHub app to **pull** the latest changes from the repo. This will download Teammate A's work and automatically merge it in with Teammate B's latest copy of things.

But wait...that sounds like it could go horribly wrong! What if Teammates A and B were editing the same file? Yes, it is important to coordinate, like so:

Good ways to collaborate:

- Teammate A can work on data cleaning in one Python script while Teammate B works on data analysis in a Jupyter notebook (a totally different file).
- Teammate A works on data analysis code (in a Python file) while Teammate B starts writing a report (in a Word doc).
- Teammate A edits code at the top of a file while Teammate B edits different code at the bottom of the same file.

If you follow one of these workflows, then you will not run into any headaches. But it is possible to create headaches in two different ways.

The first headache comes if you both edit the same part of the same file. Then when Teammate B tries to pull the changes from the repository, `git` will tell them there's a conflict and they need to resolve it. Resolving the conflict can be done, but it's a huge pain, and would probably require a trip to office hours for help. Try to avoid it. (Not that I don't want to see you in office hours—I do! But I'd love to save you the headache of the problem in the first place.)

The second headache comes if Teammate B doesn't check to be sure that Teammate A's changes integrate smoothly. Here's an example of how this might happen:

1. Becky edits the last few cells of a Jupyter notebook, sees that they work well, and commits the changes to her local repo.
2. She now wants to pass these edits to Carlo, so she uses the GitHub app to push. The app tells her she can't push yet, because Carlo pushed some changes that Becky needs to download first. This is great, because it's ensuring that the team makes sure that their work combines sensibly before publishing it online—nice!
3. So Becky clicks the Pull button in the app. Because the team was careful not to edit the same code, it works smoothly and brings Carlo's changes down to Becky's local repo on her laptop. Great!
4. At this point comes the danger: Becky can push her latest changes to the web, *but she hasn't yet checked to be sure they still work*. She knows they worked *before* she pulled Carlo's work in. But what if Carlo changed something that makes Becky's code no longer run?

It's always important, before pushing your code to the GitHub site, to check once more that it still runs correctly. If it doesn't, fix the problems and commit the fixes first, before you push to the web.

1.8.5 Complications we're skipping

Everything you need to know for using `git` in MA346 is described up above. But there is much more to `git` than this simple chapter has covered. In particular:

- We will not need to introduce the concept of “branches,” which are very important for software development teams. Branches are less important in data science than they are in software development, so we won't cover them.
- The instructions above help you avoid the concept of a “merge conflict” (when two people edit the same part of the same file). Learning how to resolve merge conflicts is an important part of `git` usage, but the instructions above should help you avoid the problem in the first place.
- There are many ways to use `git` on the command line, without the GitHub app user interface. We will not cover those in our course.

If you're a CIS major or minor and want to dive into the details we're not covering, [DataCamp has a git course](#) that covers many low-level details. Feel free to take that course if you like while you have free DataCamp access in MA346, but we won't use all those details in our work.

Learning on Your Own - VS Code's git features

If you use VS Code for your Python coding, you may find it convenient to use VS Code's git features, rather than having to switch back and forth to the GitHub app. Feel free to investigate those features on your own, and if you do so, prepare a tutorial video for the class covering:

- how to do each of the activities covered in these notes using VS Code's git support rather than the GitHub app
 - the advantages and disadvantages to each of those two options
-

1.9 Mathematics and Statistics in Python

See also the slides that summarize a portion of this content.

1.9.1 Math in Python

Having had CS230, you are surely familiar with Python's built-in math operators $+$, $-$, $*$, $/$, and $**$. You're probably also familiar with the fact that Python has a `math` module that you can use for things like trigonometry.

```
import math
math.cos( 0 )
```

```
1.0
```

I list here just a few highlights from that module that are relevant for statistical computations.

`math.exp(x)` is e^x , so the following computes e .

```
math.exp( 1 )
```

```
2.718281828459045
```

Natural logarithms are written $\ln x$ in mathematics, but just `log` in Python.

```
math.log( 10 ) # natural log of 10
```

```
2.302585092994046
```

There are some other functions useful for data work (like `math.dist()`, `math.comb()`, and `math.perm()`) coming in Python 3.8, but most Python tools (like pandas, NumPy, and SciPy) haven't yet been updated to work with Python 3.8. So I do not cover those functions here, and I recommend that you stick with Python 3.7 for now.

1.9.2 Naming mathematical variables

In programming, we almost never name variables with unhelpful names like `k` and `x`, because later readers of the code (or even ourselves reading it in two months) won't know what `k` and `x` actually do. The one exception to this is in mathematics, where it is normal to use single-letter variables, and indeed sometimes the letters matter.

Example 1: The quadratic formula is almost always written using the letters a , b , and c . Yes, names like `x_squared_coefficient`, `x_coefficient`, and `constant` are more descriptive, but they would lead to much uglier code that's not what anyone expects. Compare:

```
# not super easy to read, but not bad:
def quadratic_nice ( a, b, c ):
    return ( ( -b + ( b**2 - 4*a*c )**0.5 ) / ( 2*a ),
            ( -b - ( b**2 - 4*a*c )**0.5 ) / ( 2*a ) )

# oh my make it stop:
def quadratic_bad ( x_squared_coefficient, x_coefficient, constant ):
    return (
        ( -x_coefficient + \
          ( x_coefficient**2 - 4*x_squared_coefficient*constant )**0.5 ) \
          / ( 2*x_squared_coefficient ),
        ( -x_coefficient - \
          ( x_coefficient**2 - 4*x_squared_coefficient*constant )**0.5 ) \
          / ( 2*x_squared_coefficient )
    )

# of course both work fine:
quadratic_nice(3,-9,6), quadratic_bad(3,-9,6)
```

```
((2.0, 1.0), (2.0, 1.0))
```

But the first one is so much easier to read.

Example 2: Statistics always uses μ for the mean of a population and σ for its standard deviation. If we wrote code where we used `mean` and `standard_deviation` for those, that wouldn't be hard to read, but it wouldn't be as clear, either.

Interestingly, you can actually type Greek letters into Python code and use them as variable names! In Jupyter, just type a backslash (`\`) followed by the name of the letter (such as `mu`) and then press the Tab key. It will replace the code `\mu` with the actual letter μ . I've done so in the example code below.

```
def normal_pdf (  $\mu$ ,  $\sigma$ , x ):
    """The value of the probability density function for
    the normal distribution  $N(\mu, \sigma^2)$ , with mean  $\mu$  and
    variance  $\sigma^2$ ."""
    shifted = ( x -  $\mu$  ) /  $\sigma$ 
    return math.exp( -shifted**2 / 2.0 ) \
           / math.sqrt( 2*math.pi ) /  $\sigma$ 

normal_pdf( 10, 2, 15 )
```

```
0.00876415024678427
```

1.9.3 But what about NumPy?

Most data science projects in Python import both pandas and NumPy. Since NumPy implements tons of mathematical tools, why bother using the ones in Python's built-in `math` module? Well, on the one hand, NumPy doesn't have *everything*; for instance, the `math.comb()` and `math.perm()` functions mentioned above don't exist in NumPy. But when you *can* use NumPy, you *should*, for the following important reason.

Big Picture

All the functions in NumPy are *vectorized*, meaning that they will automatically apply themselves to every element of a NumPy array. For instance, you can just as easily compute `square(5)` (and get 25) as you can compute `square(x)` if `x` is a list of 1000 entries. NumPy notices that you provided a list of things to square, and it squares them all. What are the benefits to vectorization?

1. Using vectorization saves you *the work of writing loops*.
2. Using vectorization saves the readers of your code *the work of reading and understanding loops*.
3. If you had to write a loop to apply a Python function (like `lambda x: x**2`) to a list of 1000 entries, then the loop would (obviously) run in Python. Although Python is a very convenient language to code in, it does not produce very fast-running code. Tools like NumPy are written in languages like C++, which are less convenient to code in, but produce faster-running results. So if you can have NumPy automatically loop over your data, rather than writing a loop in Python, *the code will execute faster*.

We will return to vectorization and loops in Chapter 11 of these notes. For now, let's just run a few NumPy functions. In each case, notice that we give it an array as input, and it automatically knows that it should take action on each entry in the array.

```
# Create an array of 30 random numbers to work with.
import numpy as np
values = np.random.rand( 30 )
values
```

```
array([0.20850962, 0.31168728, 0.46599024, 0.16659761, 0.59516571,
       0.92613125, 0.13906747, 0.14225855, 0.40086385, 0.12190931,
       0.74776587, 0.27625783, 0.46294778, 0.50737946, 0.46406659,
       0.0093127 , 0.1482293 , 0.90933297, 0.87896433, 0.47943593,
       0.10416273, 0.46958161, 0.76569984, 0.09403733, 0.70498404,
       0.12742527, 0.77028858, 0.66514029, 0.5969186 , 0.53189692])
```

```
np.around( values, 2 ) # round to 2 decimal digits
```

```
array([0.21, 0.31, 0.47, 0.17, 0.6 , 0.93, 0.14, 0.14, 0.4 , 0.12, 0.75,
       0.28, 0.46, 0.51, 0.46, 0.01, 0.15, 0.91, 0.88, 0.48, 0.1 , 0.47,
       0.77, 0.09, 0.7 , 0.13, 0.77, 0.67, 0.6 , 0.53])
```

```
np.exp( values ) # compute e^x for each x in the array
```

```
array([1.23184079, 1.36572754, 1.59359145, 1.18127884, 1.81333141,
       2.52472273, 1.14920164, 1.15287468, 1.49311396, 1.12965165,
       2.11227563, 1.31818769, 1.58875037, 1.66093295, 1.59052888,
       1.0093562 , 1.1597788 , 2.48266598, 2.4084041 , 1.61516308,
       1.10978103, 1.59932491, 2.15049886, 1.09860075, 2.02381438,
       1.13589997, 2.16038961, 1.94476334, 1.81651276, 1.70215811])
```

```
np.square( values ) # square each value
```

```
array([4.34762632e-02, 9.71489629e-02, 2.17146905e-01, 2.77547639e-02,
       3.54222222e-01, 8.57719089e-01, 1.93397615e-02, 2.02374937e-02,
       1.60691824e-01, 1.48618808e-02, 5.59153790e-01, 7.63183880e-02,
       2.14320646e-01, 2.57433918e-01, 2.15357799e-01, 8.67264725e-05,
       2.19719240e-02, 8.26886456e-01, 7.72578292e-01, 2.29858811e-01,
       1.08498733e-02, 2.20506887e-01, 5.86296247e-01, 8.84301924e-03,
       4.97002492e-01, 1.62371984e-02, 5.93344498e-01, 4.42411608e-01,
       3.56311814e-01, 2.82914335e-01])
```

Notice that this makes it very easy to compute certain mathematical formulas. For example, when we want to measure the quality of a model, we might compute the RSSE, or Root Sum of Squared Errors, that is, the square root of the sum of the squared differences between each actual data value y_i and its predicted value \hat{y}_i . In math, we write it like this:

$$\text{RSSE} = \sqrt{\sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

The summation symbol lets you know that a loop will take place. But in NumPy, we can do it without writing any loops.

```
ys      = np.array( [ 1, 2, 3, 4, 5 ] ) # made up data
yhats   = np.array( [ 2, 1, 0, 3, 4 ] ) # also made up
RSSE    = np.sqrt( np.sum( np.square( ys - yhats ) ) )
RSSE
```

```
3.605551275463989
```

Notice how the NumPy code also reads just like the English: It's the square root of the sum of the squared differences; the code literally says that in the formula itself! If we had had to write it in pure Python, we would have used either a loop or a list comprehension, like in the example below.

```
RSSE = math.sqrt( sum( [ ( ys[i] - yhats[i] )**2 for i in range(len(ys)) ] ) ) # not
↳as readable
RSSE
```

```
3.605551275463989
```

A comprehensive list of NumPy's math routines appear in the [NumPy documentation](#).

1.9.4 Binding function arguments

Many functions in statistics have two types of parameters. Some of the parameters you change very rarely, and others you change all the time.

Example 1: Consider the `normal_pdf` function whose code appears earlier in this chapter. It has three parameters, μ , σ , and x . You'll probably have a particular normal distribution you want to work with, so you'll choose μ and σ , and then you'll want to use the function on many different values of x . So the first two parameters we choose just once, and the third parameter changes all the time.

Example 2: Consider fitting a linear model $\beta_0 + \beta_1 x$ to some data x_1, x_2, \dots, x_n . That linear model is technically a function of three variables; we might write it as $f(\beta_0, \beta_1, x)$. But when we fit the model to the data, then β_0 and β_1 get chosen, and we don't change them after that. But we might plug in hundreds or even thousands of different x values to f , using the same β_0 and β_1 values each time.

Programmers have a word for this; they call it *binding* the arguments of a function. Binding allows us to tell Python that we've chosen values for some parameters and won't be changing them; Python can thus give us a function with fewer parameters, to make things simpler. Python does this with a tool called `partial` in its `functools` module. Here's how we would apply it to the `normal_pdf` function.

```
from functools import partial

# Let's say I want the standard normal distribution, that is,
# I want to fill in the values  $\mu=0$  and  $\sigma=1$  once for all.
my_pdf = partial( normal_pdf, 0, 1 )

# now I can use that on as many x inputs as I like, such as:
my_pdf( 0 ), my_pdf( 1 ), my_pdf( 2 ), my_pdf( 3 ), my_pdf( 4 )
```

```
(0.3989422804014327,
 0.24197072451914337,
 0.05399096651318806,
 0.0044318484119380075,
 0.00013383022576488537)
```

In fact, SciPy's built-in random number generating procedures let you use them either by binding arguments or not, at your preference. For instance, to generate 10 random floating point values between 0 and 100, we can do the following. (The `rvs` function stands for "random values.")

```
import scipy.stats as stats
stats.uniform.rvs( 0, 100, size=10 )
```

```
array([76.85768748, 48.01593333, 92.85262834, 41.81845963, 26.91906569,
       34.43715542, 22.2555      , 44.79309438, 20.67422863, 38.03269724])
```

Or we can use built-in SciPy functionality to bind the first two arguments and create a specific random variable, then call `rvs` on that.

```
X = stats.uniform( 0, 100 ) # make a random variable
X.rvs( size=10 )           # generate 10 values from it
```

```
array([94.36173421, 37.41078276, 49.82941889, 31.62948686, 33.37754507,
       5.79206238, 71.32878629, 48.11541087, 99.20472891, 40.9434339 ])
```

The same random variable can, of course, be used to create more values later.

The `partial` tool built into Python only works if you want to bind the *first* arguments of the function. If you need to bind later ones, then you can do it yourself using a `lambda`, as in the following example.

```
def subtract ( a, b ): # silly little example function
    return a - b

subtract_1 = lambda a: subtract( a, 1 ) # bind second argument to 1

subtract_1( 5 )
```

```
4
```

We will also use the concept of binding function parameters when we come to curve fitting at the end of this chapter.

1.9.5 GB213 in Python

You can refer at any time to one of the appendices in these course notes, a *review of GB213, but in Python*.

Topics covered there:

- Discrete and continuous random variables
 - creating
 - plotting
 - generating random values
 - computing probabilities
 - computing statistics
- Hypothesis testing for a population mean
 - one-sided
 - two-sided
- Simple linear regression (one predictor variable)
 - creating the model from data
 - computing R and R^2
 - visualizing the model

Topics not covered in that chapter, but that you may have seen in GB213:

- Basic probability (covered in every GB213 section)
- ANOVA (covered in some GB213 sections)
- χ^2 tests (covered in some GB213 sections)

Learning on Your Own - Pingouin

The GB213 review appendix that I linked to above uses the very popular Python statistics tools `statsmodels` and `scipy.stats`. But there is a relatively new toolkit called Pingouin; it's not as popular (yet?) but it has some advantages over the other two. See [this blog post](#) for an introduction and consider a tutorial, video, presentation, or notebook for the class that showcases when you might prefer Pingouin to the others, and how to use it in such cases. Be sure to include the installation procedure.

1.9.6 Curve fitting in general

The final topic covered in the GB213 review mentioned above is simple linear regression, which fits a line to a set of (two-dimensional) data points. But Python's scientific tools permit you to handle much more complex models. We cannot cover mathematical modeling in detail in MA346, because it can take several courses on its own, but you can learn more about regression modeling in particular in [MA252 at Bentley](#). But we will cover how to fit an arbitrary curve to data in Python.

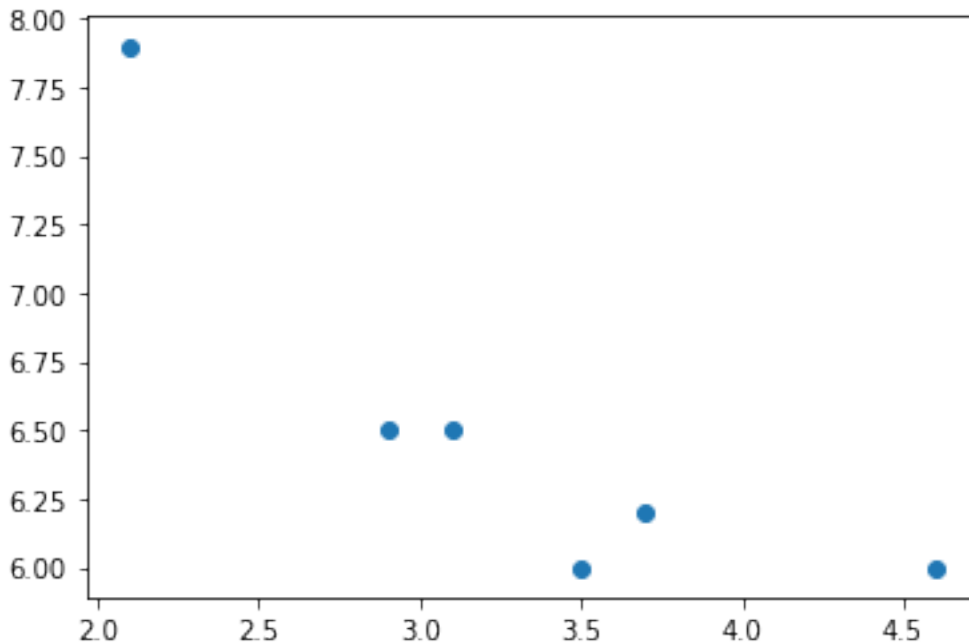
1. Say we have some data

We will assume you have data stored in a pandas DataFrame, and we will lift out just two columns of the DataFrame, one that will be used as our x values (independent variable), and the other as our y values (dependent variable). I'll make up some data here just for use in this example.

```
# example data only, totally made up:
import pandas as pd
df = pd.DataFrame( {
    'salt used (x)' :    [ 2.1, 2.9, 3.1, 3.5, 3.7, 4.6 ],
    'ice remaining (y)' : [ 7.9, 6.5, 6.5, 6.0, 6.2, 6.0 ]
} )
df
```

	salt used (x)	ice remaining (y)
0	2.1	7.9
1	2.9	6.5
2	3.1	6.5
3	3.5	6.0
4	3.7	6.2
5	4.6	6.0

```
import matplotlib.pyplot as plt
xs = df['salt used (x)']
ys = df['ice remaining (y)']
plt.scatter( xs, ys )
plt.show()
```



2. Choose a model

Curve-fitting is a powerful tool, and it's easy to misuse it by fitting to your data a model that doesn't make sense for that data. A mathematical modeling course can help you learn how to assess the appropriateness of a given type of line, curve, or more complex model for a given situation. But for this small example, let's pretend that we know that the following model makes sense, perhaps because some earlier work with salt and ice had success with it. (Again, keep in mind that this example is really, truly, totally made up.)

$$y = \frac{\beta_0}{\beta_1 + x} + \beta_2$$

We will use this model. Obviously, it's not the equation of a line, so linear regression tools like those covered in the GB213 review notebook won't be sufficient. To begin, we code the model as a Python function taking inputs in this order: first, x , then after it, all the model parameters β_0, β_1 , and so on, however many model parameters there happen to be (in this case three).

```
def my_model ( x,  $\beta_0$ ,  $\beta_1$ ,  $\beta_2$  ) :
    return  $\beta_0$  / (  $\beta_1$  + x ) +  $\beta_2$ 
```

3. Have SciPy find the β s

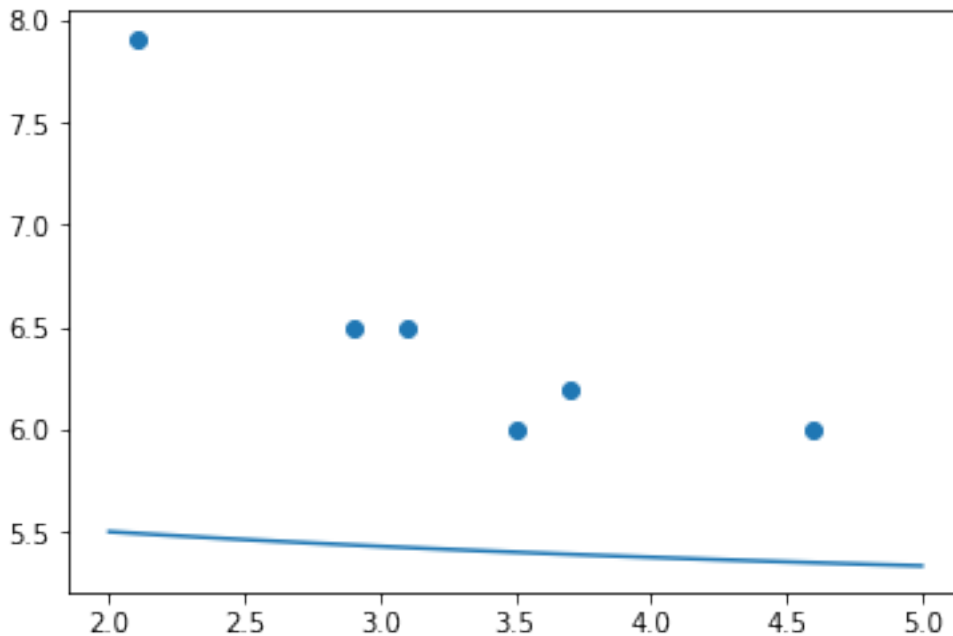
This step is called “fitting the model to your data.” It finds the values of $\beta_0, \beta_1, \beta_2$ that make the most sense for the particular x and y data values that you have. Using the language from earlier in this chapter, SciPy will tell us how to *bind values to the parameters* $\beta_0, \beta_1, \beta_2$ of `my_model` so that the resulting function, which just takes x as input, is the one best fit to our data.

For example, if we picked our own values for the model parameters, we would probably guess poorly. Let's try guessing $\beta_0 = 1, \beta_1 = 2, \beta_2 = 3$.

```
guess_model = lambda x: my_model( x, 3, 4, 5 )

import numpy as np
many_xs = np.linspace( 2, 5, 100 )

plt.scatter( xs, ys )
plt.plot( many_xs, guess_model( many_xs ) )
plt.show()
```



Yyyyyyeah... Our model is nowhere near the data. That's why we need SciPy to find the β s. Here's how we ask it to do so. You start with your own guess for the parameters, and SciPy will improve it.

```
from scipy.optimize import curve_fit
my_guessed_betas = [ 3, 4, 5 ]
found_betas, covariance = curve_fit( my_model, xs, ys, p0=my_guessed_betas )
β0, β1, β2 = found_betas
β0, β1, β2
```

```
(1.3739384272240622, -1.5255461192343747, 5.510233385761209)
```

So how does SciPy's found model look?

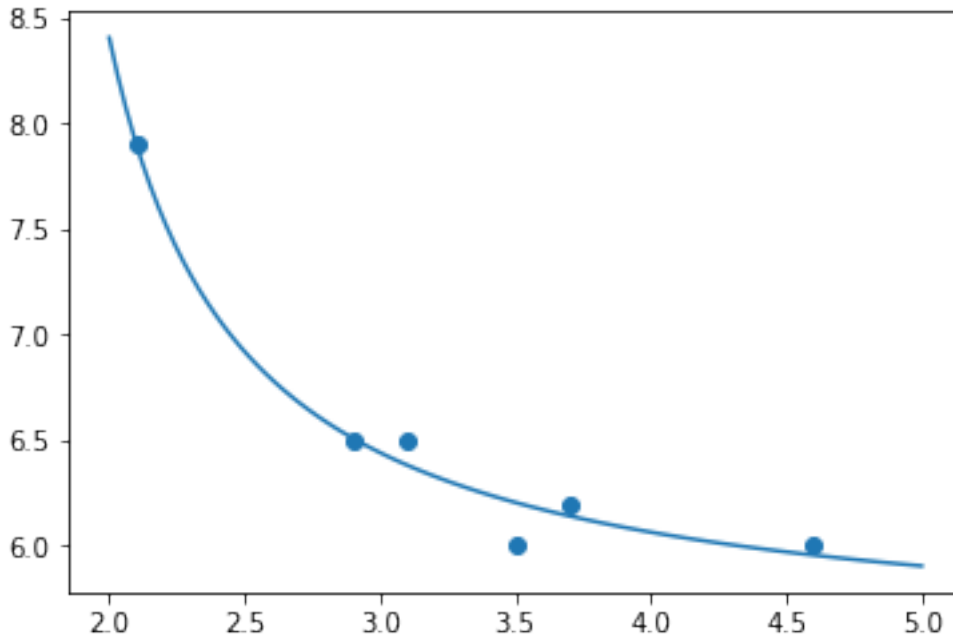
4. Describe and show the fit model

Rounding to a few decimal places, our model is therefore the following:

$$y = \frac{1.37}{-1.53 + x} + 5.51$$

It fits the data very well, as you can see below.

```
fit_model = lambda x: my_model( x, β0, β1, β2 )
plt.scatter( xs, ys )
plt.plot( many_xs, fit_model( many_xs ) )
plt.show()
```

Big Picture

In mathematical modeling and machine learning, we sometimes distinguish between a *model* and a *fit model*.

- A *model* is a general purpose technique that you decide might suit the data. Examples:
 - A linear model, $y = \beta_0 + \beta_1 x$
 - A quadratic model, $y = \beta_0 + \beta_1 x + \beta_2 x^2$
 - A logistic curve, $y = \frac{\beta_0}{1 + e^{\beta_1(-x + \beta_2)}}$
 - A neural network
- A *fit model* is the specific version of the general model that's been tailored to suit your data. We create it from the general model by *binding* the values of the β s to specific numbers.

For example, if your model were $y = \beta_0 + \beta_1 x$, then your fit model might be $y = -0.95 + 1.13x$. In the general model, y depends on three variables (x, β_0, β_1). In the fit model, it depends on only one variable (x). So model fitting is an example of binding the variables of a function.

In class, we will use this technique to fit a logistic growth model to COVID-19 data. Be sure to have completed the preparatory work on writing a function that extracts the series of COVID-19 cases over time for a given state! Recall that it appears on the final slide of the Chapter 8 slides.

1.10 Visualization

See also the slides that summarize a portion of this content.

In preparation for today, you learned many [data visualization tools from DataCamp](#). In fact, if you're doing this reading before you do the DataCamp homework, I strongly suggest that you stop here, do the DataCamp first, and then come back here.

Rather than review those tools here, I will categorize them instead. This page is therefore a reference in which you can look up the kind of data you *have* and see which visualizations make the most sense for it, and what each one accomplishes.

We will use two datasets throughout the examples below. The first is a set of sales data for the employees of an imaginary company (Dunder Mifflin, perhaps?). The data has the following format, organized by employee ID numbers, and including year, quarter, sales quantity, and bonus earned for each ID in each relevant time frame.

```
import pandas as pd
sales_df = pd.read_csv( './_static/fictitious-sales-data.csv' )
sales_df.head()
```

	emp_id	year	quarter	sales	bonus
0	1275342	2010	2	8.000000	0
1	1275342	2010	3	333.000000	0
2	1275342	2010	4	594.000000	2000
3	1275342	2011	1	276.066177	0
4	1275342	2011	2	340.000000	0

The second dataset is the basic NASDAQ data for Renewable Energy Group, Inc. (symbol REGI) for the first half of 2020.

```
regi_df = pd.read_csv( './_static/regi-prices-2020.csv' )
regi_df.head()
```

	Date	Open	High	Low	Close	Adj Close	Volume
0	2-Jan-20	27.21	27.95	26.62	27.89	27.89	781100
1	3-Jan-20	28.16	28.95	27.73	28.82	28.82	1405100
2	6-Jan-20	28.53	28.81	28.00	28.39	28.39	716800
3	7-Jan-20	28.17	28.28	26.08	26.44	26.44	1378900
4	8-Jan-20	26.37	26.40	24.86	25.19	25.19	1195900

1.10.1 What if I have two columns of numeric data?

This situation is *extremely common*, and that's why we address it first. If we consider the two datasets described above, we can find many ways to create two columns of numeric data, including the following examples.

1. The year and sales columns from `sales_df`
2. The year and sales columns we would get by grouping `sales_df` by year
3. The Volume and High columns from `regi_df`
4. The index and the Close column from `regi_df`

Big Picture

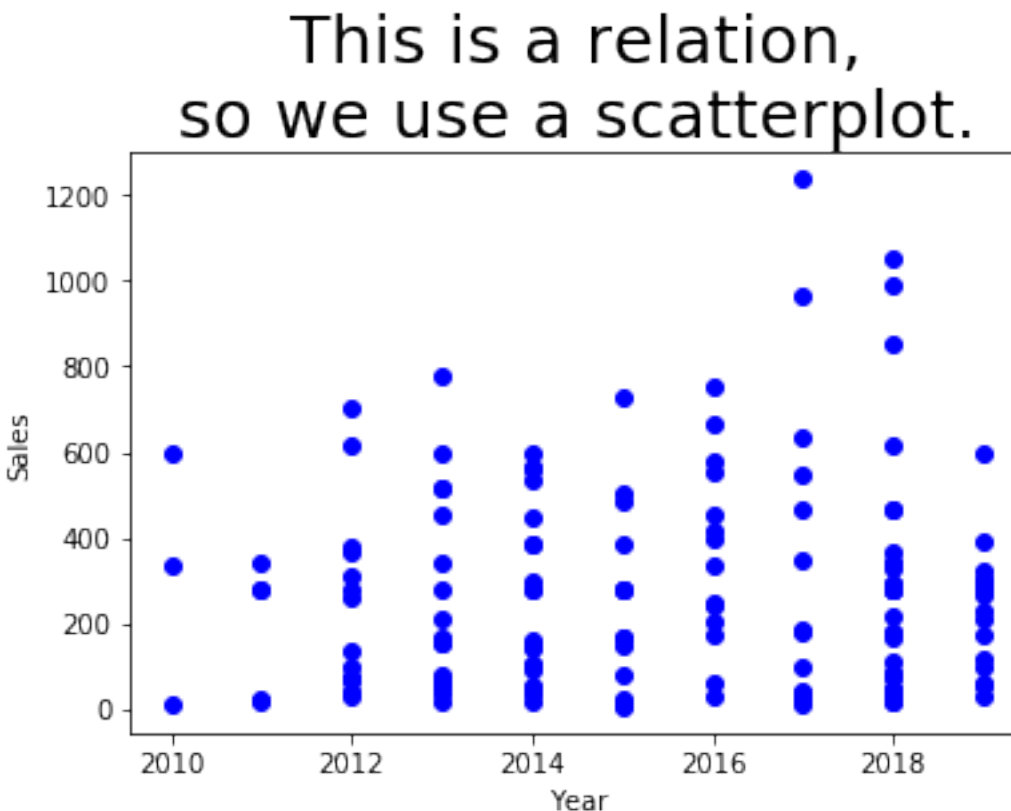
Recall that two columns of data always form a binary relation, but may or may not be a function. Noticing whether the data are a function is very important when deciding how to visualize them.

- A **function** can be shown with a **line plot**, as in algebra classes.
- A **relation** that is not a function must be shown as a **scatterplot**.

Both scatterplots and line plots are drawn with `plt.plot()` in Matplotlib. There are many ways to specify the plot type, as you've seen in DataCamp. Let's look at the same four examples mentioned above.

Example 1: The year and sales columns from `sales_df` do not form a function, because each year has multiple sales figures. We can see this if we visualize them with a scatterplot.

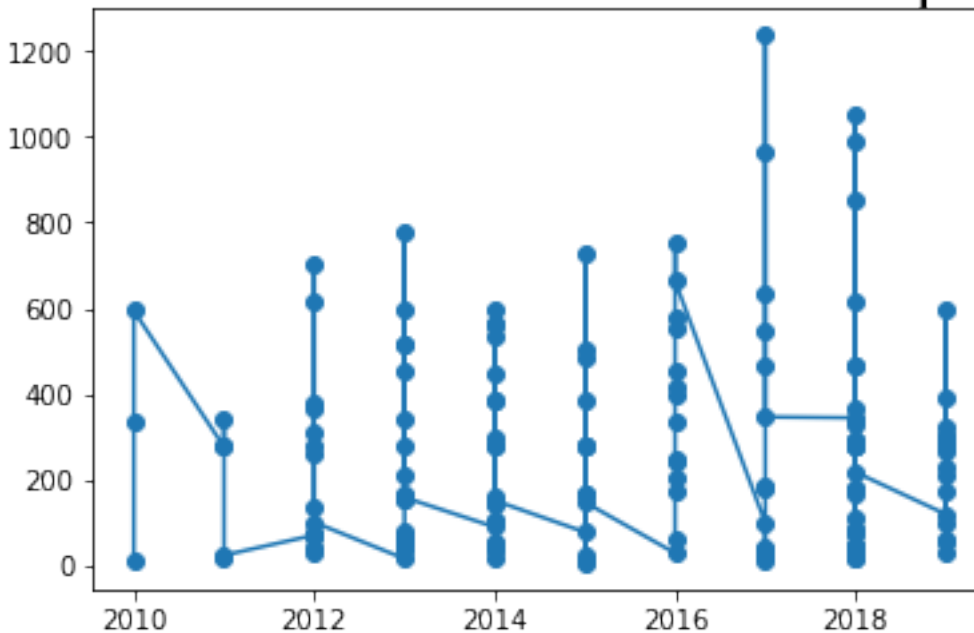
```
import matplotlib.pyplot as plt
plt.plot( sales_df['year'], sales_df['sales'], 'bo' ) # blue circles
plt.title( 'This is a relation,\nso we use a scatterplot.', fontdict={ "fontsize": 25,
↪ } )
plt.xlabel( 'Year' )
plt.ylabel( 'Sales' )
plt.show()
```



The same example would have gone quite wrong if we had attempted to use a line plot instead, as you can see below. Matplotlib tries to connect the dots in sequence to show a line, but it doesn't make any sense, because the data is not a function.

```
plt.plot( sales_df['year'], sales_df['sales'], '-o' ) # dots and lines
plt.title( 'This is a relation, so we\nshould have used a scatterplot!', fontdict={
↪ "fontsize": 25 } )
plt.show()
```

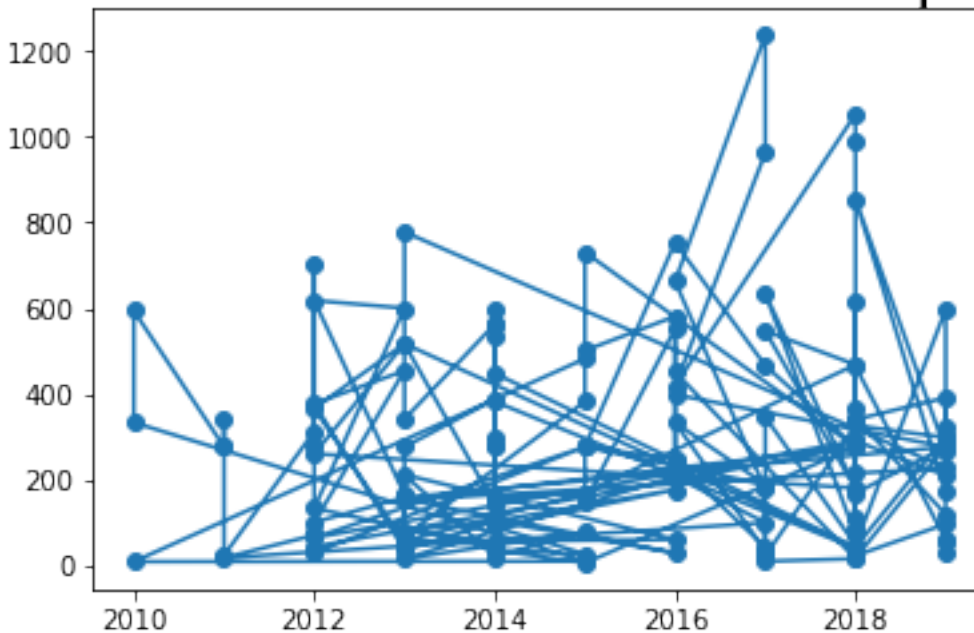
This is a relation, so we should have used a scatterplot!



It would have been even more hideous if the data hadn't been sorted by year. Let's see what it would have been like if it had been sorted by employee instead, for instance.

```
temp_df = sales_df.sort_values( 'emp_id' )
plt.plot( temp_df['year'], temp_df['sales'], '-o' ) # dots and lines
plt.title( 'This is a relation, so we\nshould have used a scatterplot!', fontdict={
    ↪ "fontsize": 25 } )
plt.show()
```

This is a relation, so we should have used a scatterplot!

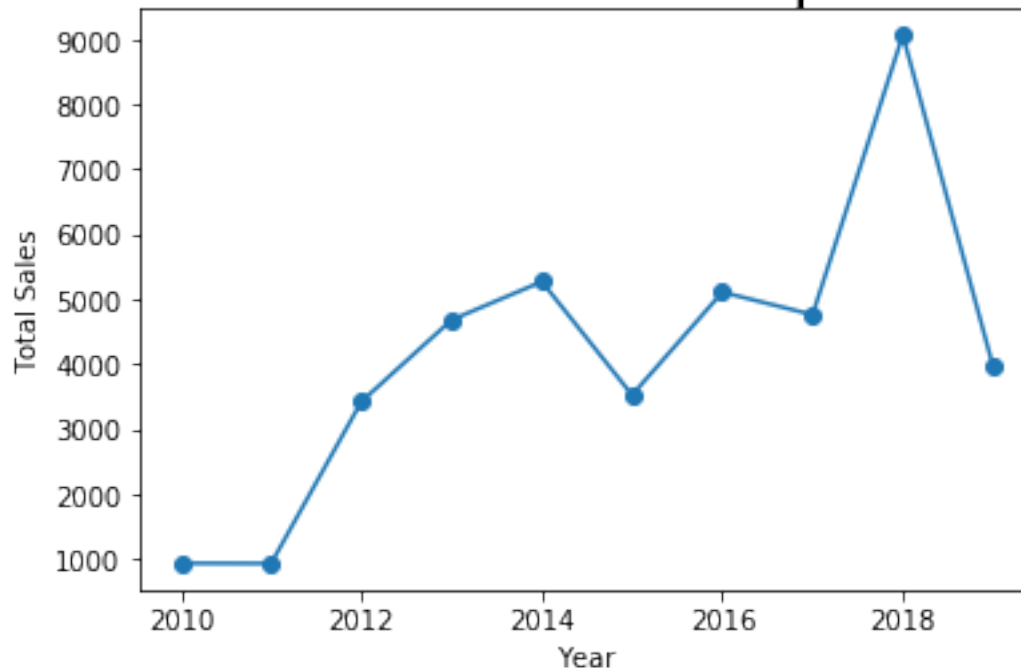


The bad graphs just shown illustrate the importance of knowing whether your data is a function or relation, and choosing the appropriate plotting technique. Let's see how line plots can look nice when the data *is* a function.

Example 2: If we group the sales data by year, then each year appears only once, and the relationship between year and sales becomes a function. Let's use `sum()` to do the grouping, so that we can see total sales by year.

```
grouped_df = sales_df.groupby( 'year' ).sum()
plt.plot( grouped_df.index, grouped_df['sales'], '-o' ) # dots and lines
plt.title( 'This is a function,\nso we use a line plot.', fontdict={ "fontsize": 25 } )
plt.xlabel( 'Year' )
plt.ylabel( 'Total Sales' )
plt.show()
```

This is a function,
so we use a line plot.



That plot looks the way we expect. It is especially sensible because the independent variable (x axis) is sequential, so it makes sense for us to think of the data as connected and flowing from left to right.

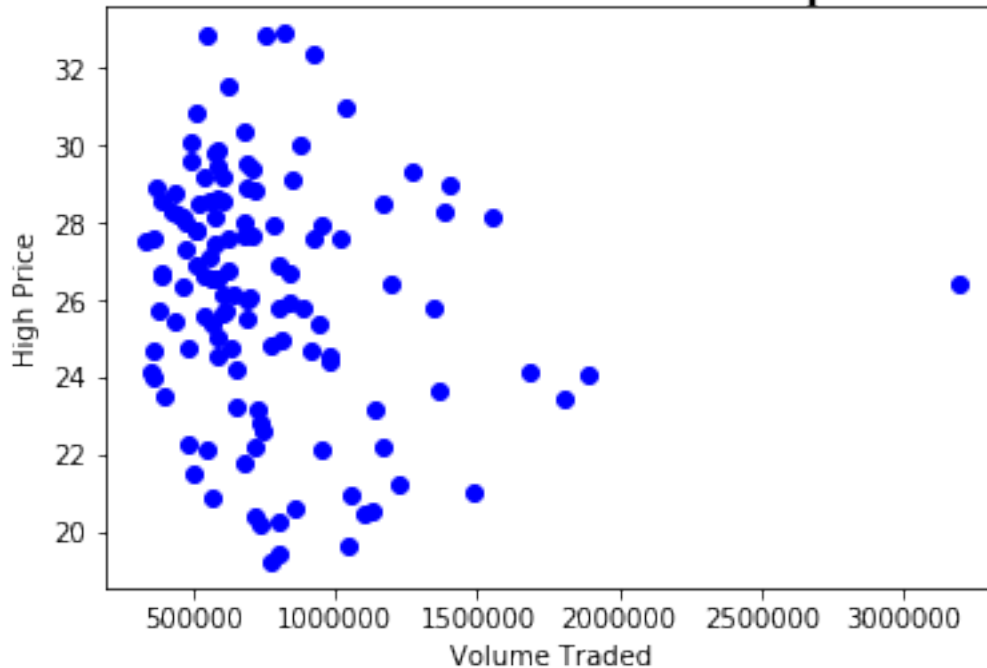
Note that if your data aren't already sorted by the independent variable, connecting the dots with lines will jump all over your plot as it plots points in the wrong order. Use `sort_values()` to get the data in the right order, in such a case.

Let's consider one more example of a function and a non-function, but we'll do them quickly.

Example 3: The Volume and High columns from `regi_df` may or may not be a function; it depends on the data we happened to get. The *meanings* of the columns indicate that they probably are not a function, if given enough historical data. So we'll use a scatterplot.

```
plt.plot( regi_df['Volume'], regi_df['High'], 'bo' ) # blue circles
plt.title( 'This is a relation,\nso we use a scatterplot.', fontdict={ "fontsize": 25,
↪ } )
plt.xlabel( 'Volume Traded' )
plt.ylabel( 'High Price' )
plt.show()
```

This is a relation,
so we use a scatterplot.

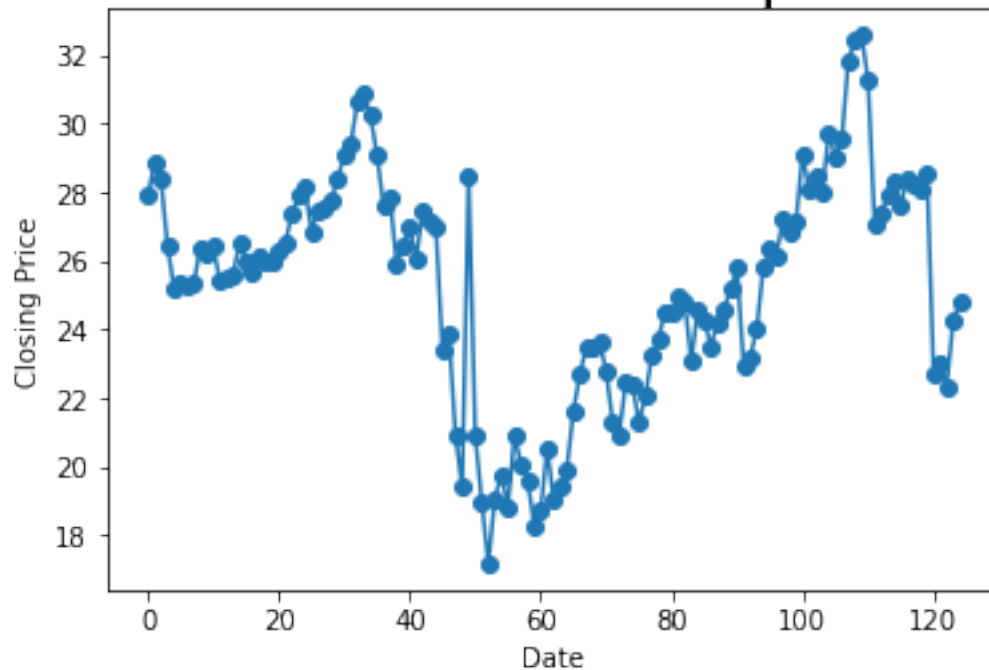


We can clearly see that there *might* be a collision in there of two x values having the same y value. Even if they don't, we certainly wouldn't want to try connecting those dots with lines; it would be a meaningless mess.

Example 4: The index and the Close column from `regi_df` are a function, because each index represents a separate day, and thus only appears once in the data. Let's see.

```
plt.plot( regi_df.index, regi_df['Close'], '-o' ) # dots and lines
plt.title( 'This is a function,\nso we use a line plot.', fontdict={ "fontsize": 25 } )
plt.xlabel( 'Date' )
plt.ylabel( 'Closing Price' )
plt.show()
```

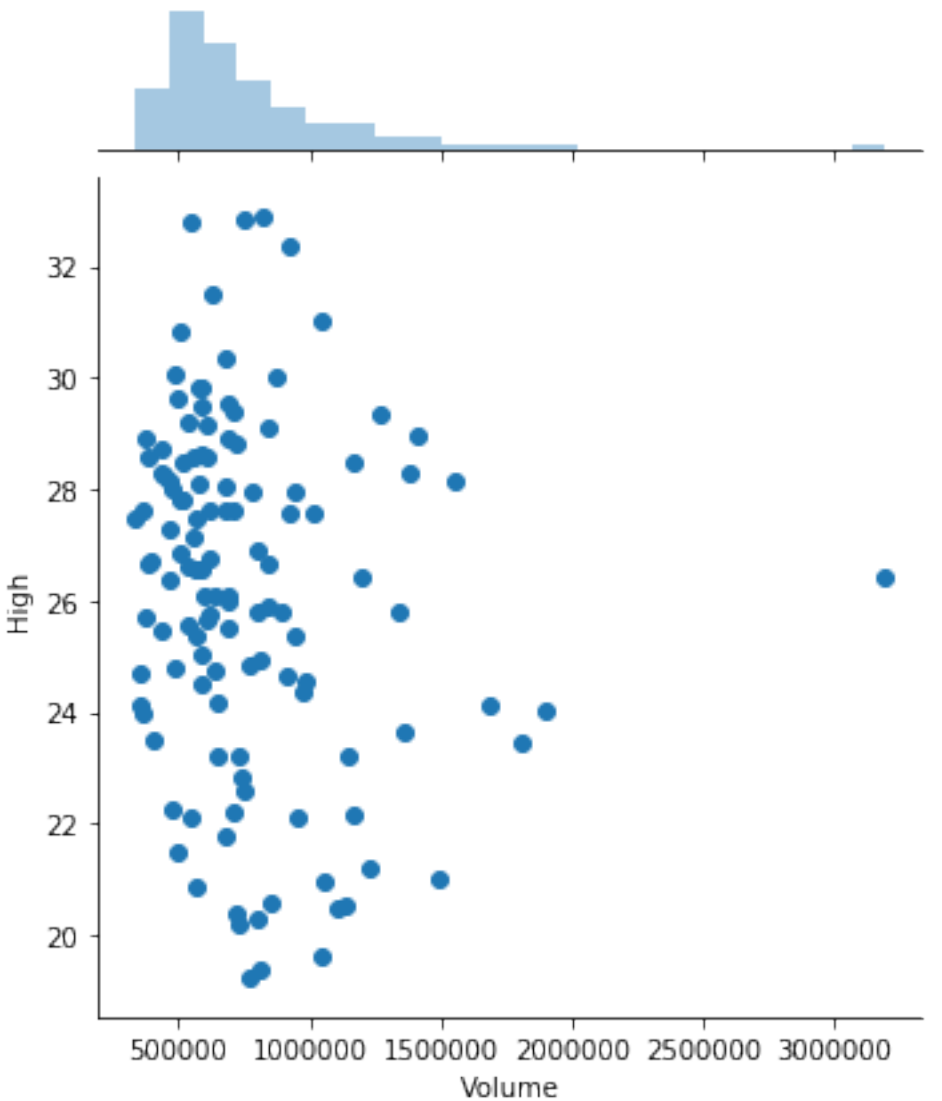
This is a function,
so we use a line plot.



1.10.2 But can my two columns of data look more awesome?

Recall that the Seaborn library makes it easy to add histograms to both the horizontal and vertical axes of a standard plot to get a better sense of the distribution. This is possible with both line and scatter plots, but it is more commonly useful with scatterplots.

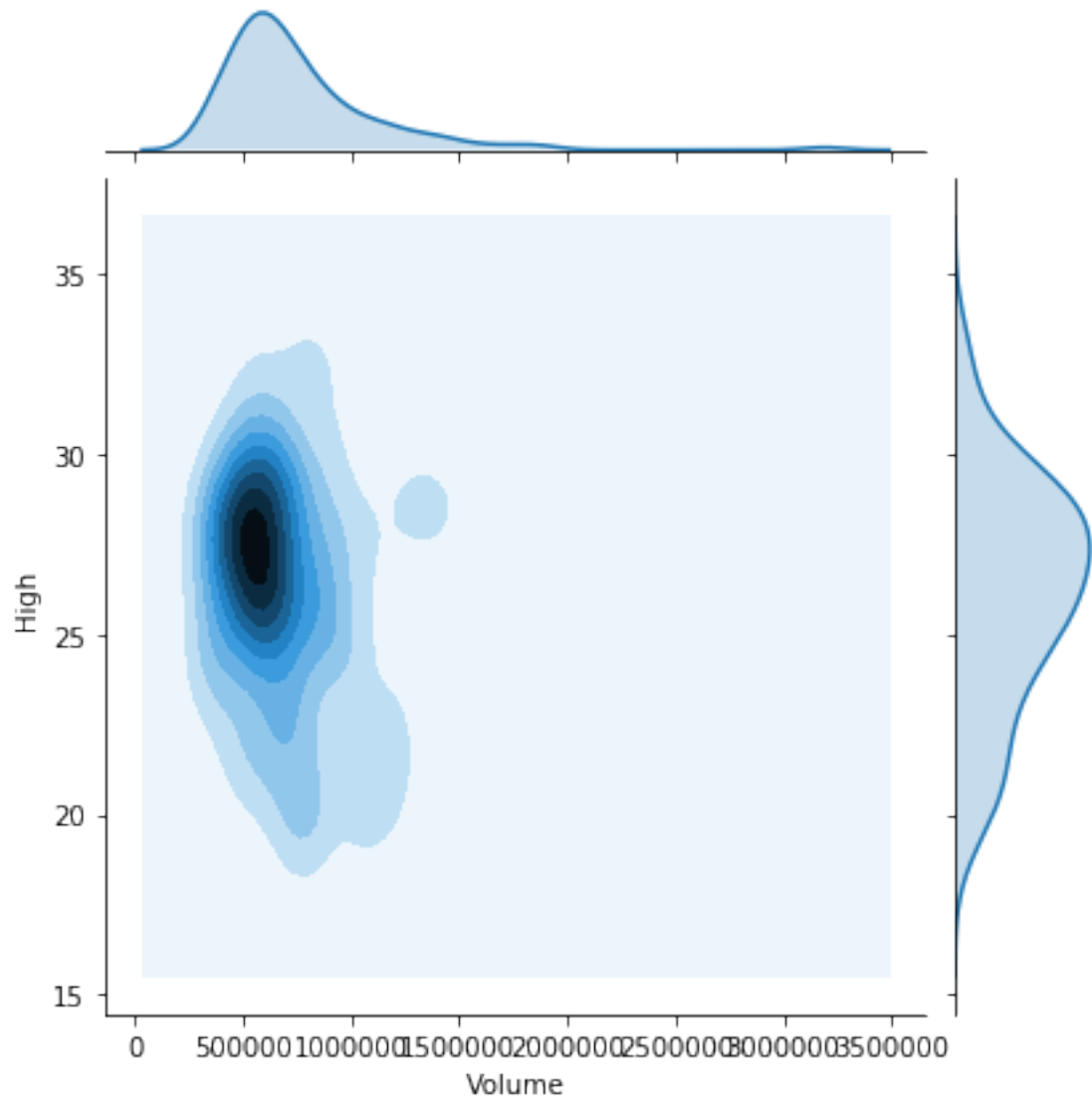
```
import seaborn as sns
sns.jointplot( x='Volume', y='High', data=regi_df )
plt.show()
```

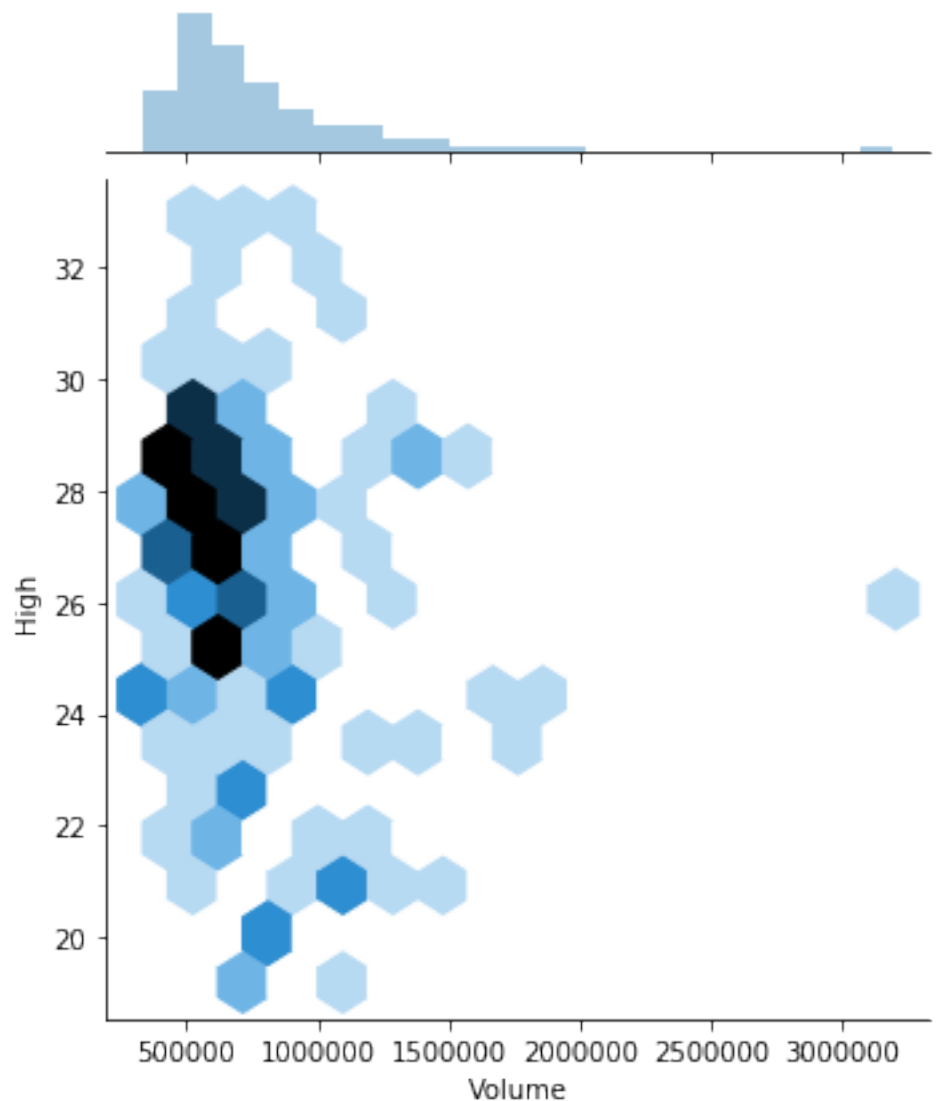
If there were thousands of datapoints (or more), I suggest trying any of the following options. I'll illustrate some of them using the same data we just saw, even though it doesn't have thousands of points.

1. Use `kind='kde'` in a joint plot to smooth the histograms, as shown in the first plot below.
2. Use `alpha=0.5` or an even smaller number, so that points in your scatterplot that stack up on top of one another show different levels of density throughout the graph.
3. Use `kind='hex'` to bin values within the scatterplot as well, again showing the varying density throughout the plot, as in the second plot below.

```
sns.jointplot( x='Volume', y='High', data=regi_df, kind='kde' )
plt.show()
```



```
sns.jointplot( x='Volume', y='High', data=regi_df, kind='hex' )  
plt.show()
```



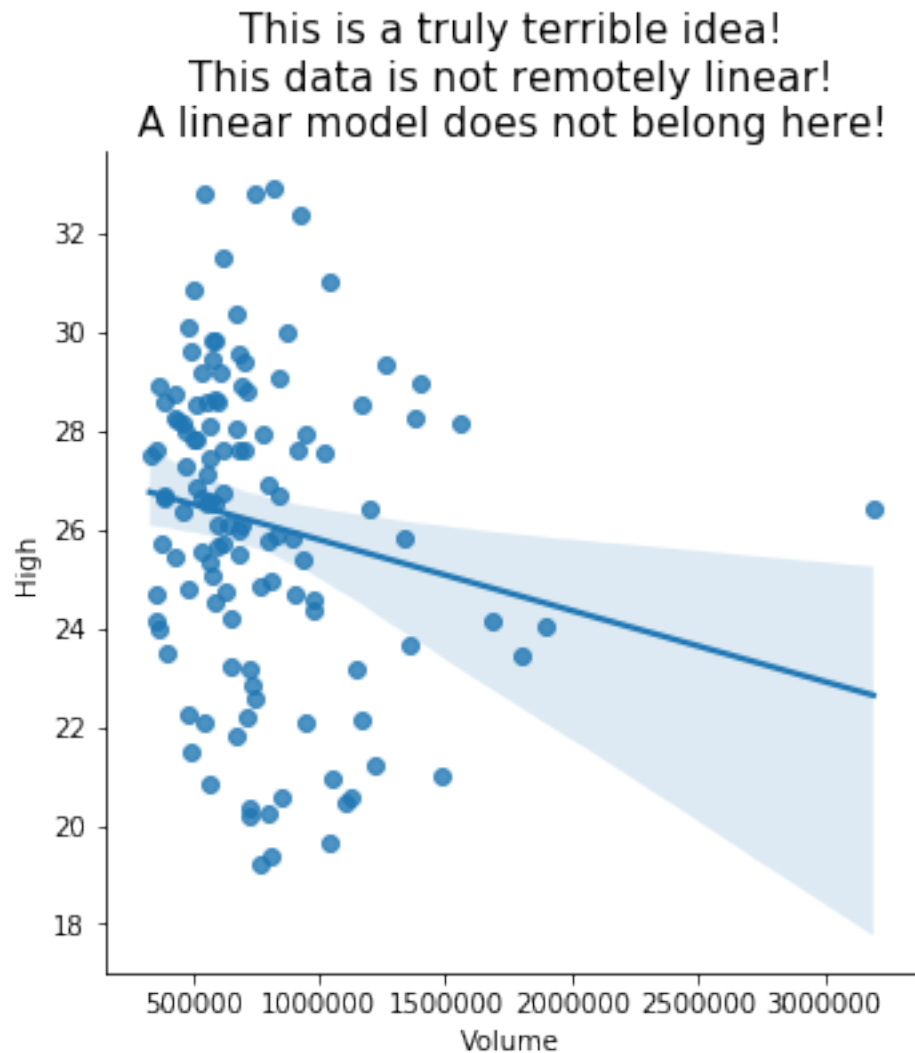
1.10.3 What if my two columns are very related?

Seaborn provides a few tools for showing how one variable depends on another.

First, you can plot a line of best fit over a scatterplot, together with confidence bars for the predictions made by that linear model. Recall from GB213 that it is not always sensible to fit a linear model to data. But in cases where it makes sense, Seaborn makes it easy to visualize.

Keep in mind that Seaborn is quite happy to show you a linear model even when it does not make any sense to do so! Just because Python will plot it for you does not mean that you should ask it to! Here's an example of just such a situation.

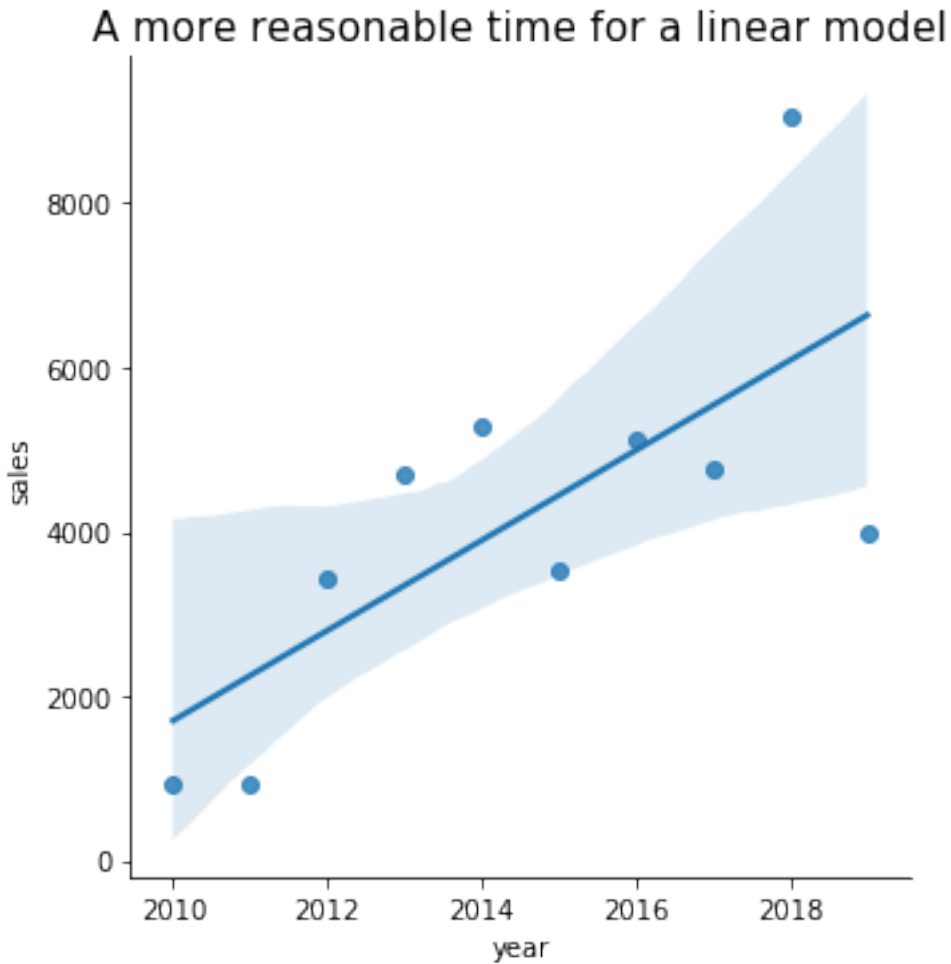
```
sns.lmplot( x='Volume', y='High', data=regi_df )
plt.title( 'This is a truly terrible idea!\n'
          + 'This data is not remotely linear!\n'
          + 'A linear model does not belong here!',
          fontdict={ "fontsize": 15 } )
plt.show()
```



Seaborn won't show you the coefficients of the model, nor measure its goodness of fit; see [the GB213 review](#) for how to do those things in Python.

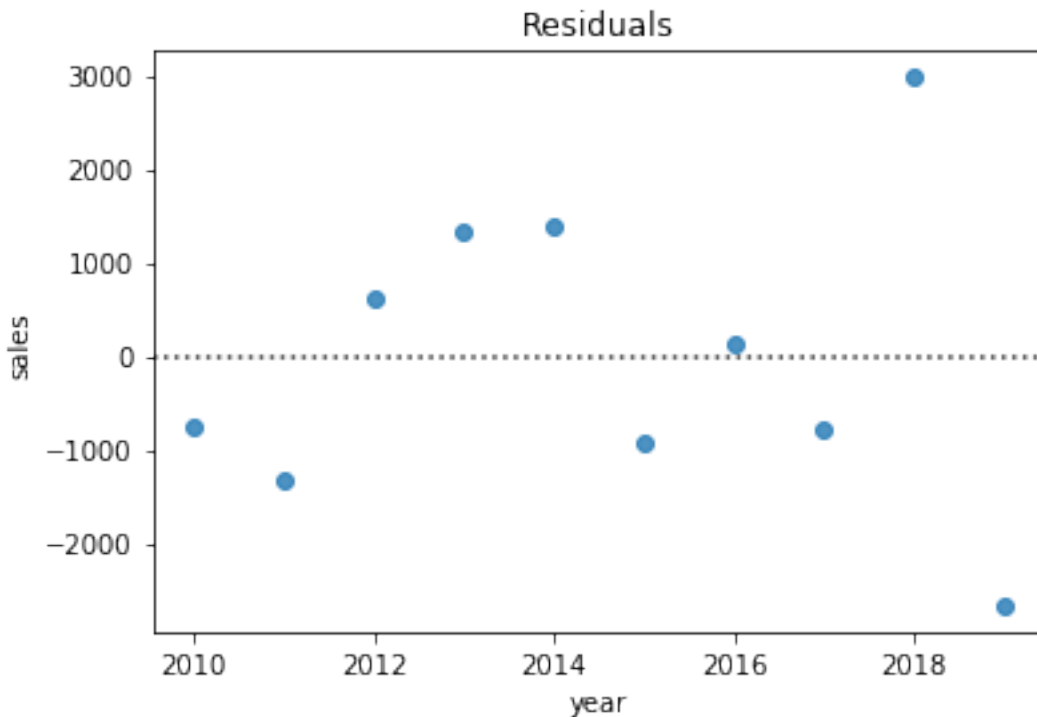
Of course, there are some situations where a linear model is reasonable, like the total sales over time plot from earlier. Seaborn is fussy about using column names only in `lmplot`, so we have to move the index in as an actual column here.

```
grouped_df['year'] = grouped_df.index
sns.lmplot( x='year', y='sales', data=grouped_df )
plt.title( 'A more reasonable time for a linear model',
           fontdict={ "fontsize": 15 } )
plt.show()
```



As you know from GB213, part of assessing whether linear regression is appropriate involves inspecting the residuals (the difference between each data point and the linear model). Seaborn makes this easy, too.

```
sns.residplot( x='year', y='sales', data=grouped_df )  
plt.title( 'Residuals' )  
plt.show()
```



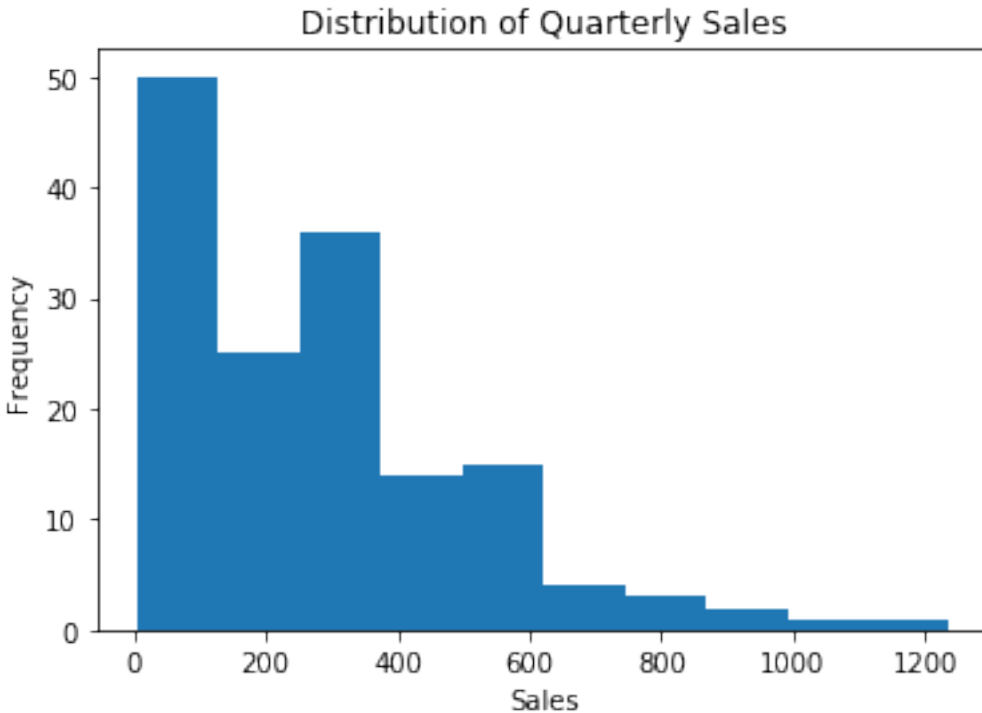
1.10.4 What if I have only one column of data?

The primary visualization tools appropriate for such a situation are variations on the idea of a histogram. These include a standard histogram plus swarm plots, strip plots, and violin plots. A secondary visualization in this situation is an ECDF, which we will return to below.

We can plot a standard histogram with `plt.hist()`, but this doesn't work very well for very small data sets. It can also suffer from “binning bias,” which distorts the actual distribution through the approximation inherent in clustering points into bars. But with many data points distributed smoothly along the horizontal axis, it often works well.

When labeling a histogram, the y axis is almost always “frequency” and the title should typically mention the idea of a “distribution.”

```
plt.hist( sales_df['sales'] )
plt.xlabel( 'Sales' )
plt.ylabel( 'Frequency' )
plt.title( 'Distribution of Quarterly Sales' )
plt.show()
```

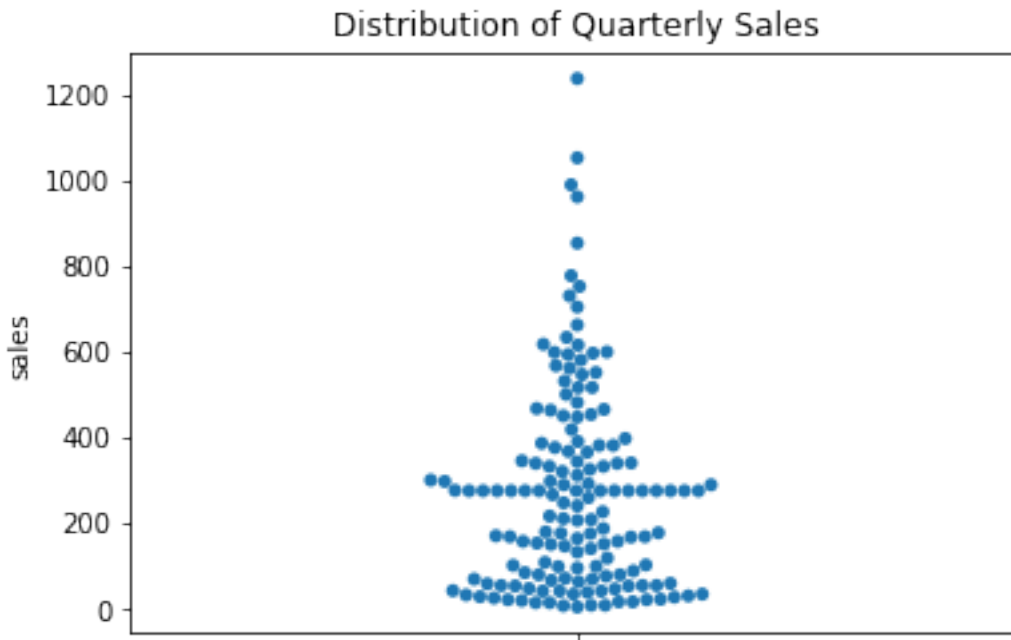


Matplotlib's built-in `plt.hist()` works fine, but to upgrade your histogram game, consider checking out [Seaborn's `sns.distplot\(\)`](#), which also shows histograms, but with handy options for commonly-desired additional features.

To remove the problem of binning bias, you can try a swarm plot. This works well with a small-to-medium number of data points, but becomes unmanageable for large datasets, because it attempts to give each data point its own visual space. Also, data points are just plotted *close* to where they actually belong, so the distortion of a histogram's binning bias has been reduced, but not fully removed. The picture is still an approximation of the actual data, but still much more accurate than a histogram.

Note that in a one-column swarm plot, there is no horizontal variable, and thus we do not label that axis.

```
sns.swarmplot( y='sales', data=sales_df )
plt.title( 'Distribution of Quarterly Sales' )
plt.show()
```

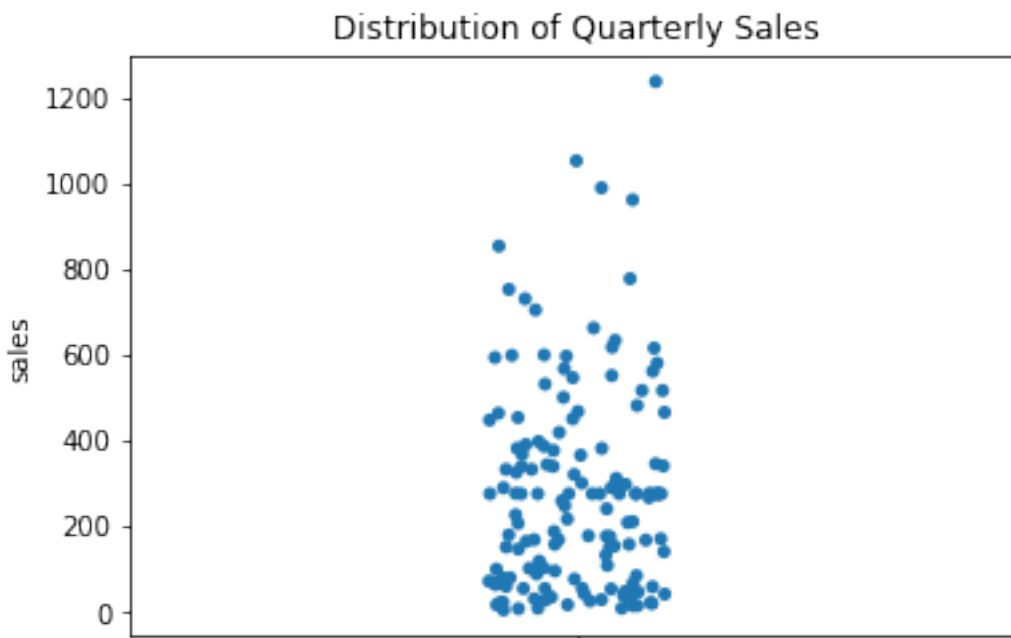


A swarm plot can get quite wide if there are many data points clustered in a small area. If your data has this problem, try using a strip plot, which keeps a constant width everywhere.

This comes at a price, however. Some data points are stacked on top of one another, so you won't really be able to see as much variation in density. You can combat this problem by choosing `alpha=0.5` or some smaller number, so that overlapping data points show variations in color.

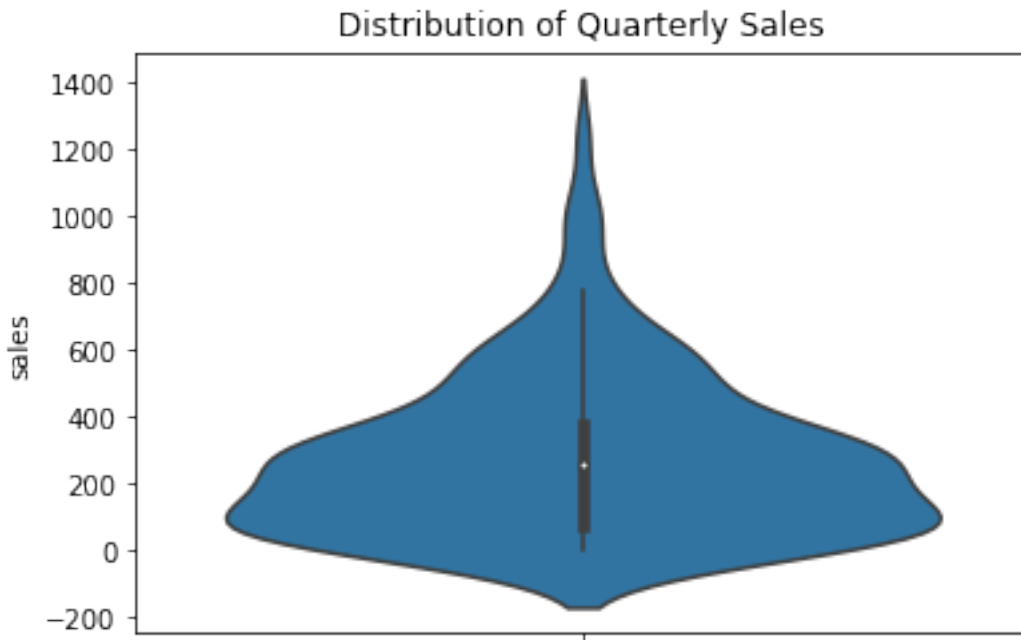
Finally, a strip plot uses random jittering to place the points, so it won't always look the same each time you render it!

```
sns.stripplot( y='sales', data=sales_df )  
plt.title( 'Distribution of Quarterly Sales' )  
plt.show()
```



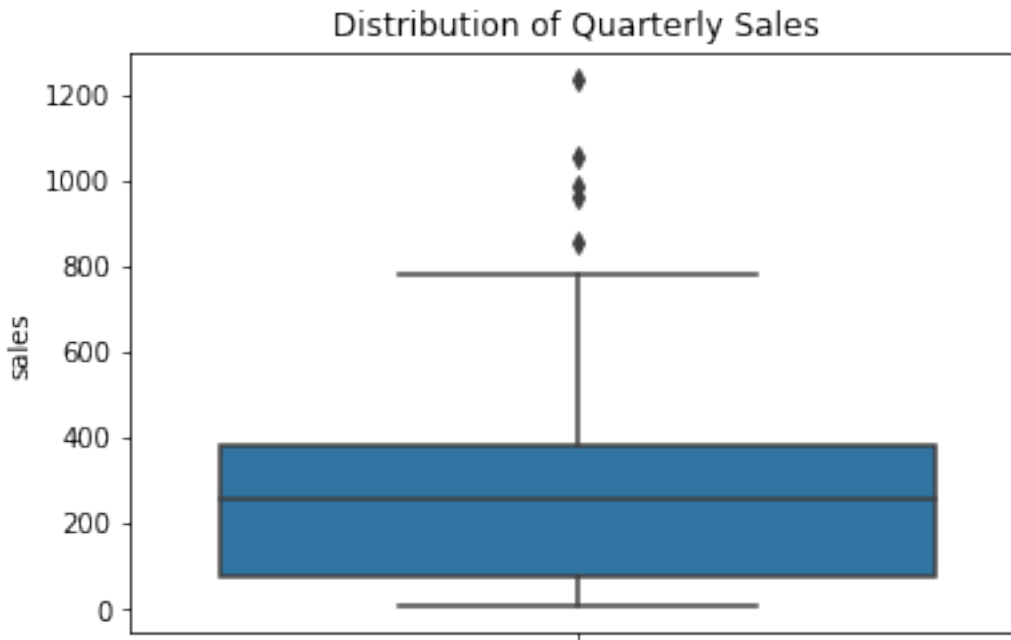
Lastly, if you have enough data, you may want to simply smooth it out into curves instead. This is not a faithful representation of sparse data, but it can be a faithful representation of a very large dataset.

```
sns.violinplot( y='sales', data=sales_df )  
plt.title( 'Distribution of Quarterly Sales' )  
plt.show()
```



Finally, if you care only about the quartiles of the distribution (25%, 50%, 75%) and the outliers, you can use a box plot.

```
sns.boxplot( y='sales', data=sales_df )  
plt.title( 'Distribution of Quarterly Sales' )  
plt.show()
```



Every one of the options above can also be shown horizontally instead. Just use `orient='h'` in the plotting command.

```
sns.swarmplot( y='sales', data=sales_df, orient='h' )  
plt.title( 'Distribution of Quarterly Sales' )  
plt.show()
```



1.10.5 Can't I test a single column for normality?

I'm so glad you asked! One of the most common assumptions in statistics is that a dataset comes from an approximately normally distributed population. We can get a sense of whether that holds true for some dataset we have by plotting the cumulative distribution function (CDF) of the data against that of a normal distribution, as you saw in DataCamp. (A CDF from data is called an empirical CDF, or ECDF.)

While DataCamp did it manually, there are libraries that can handle it for you. *The notes for Chapter 9* suggested a Learning On Your Own activity about Pingouin, a new Python statistics module, which implements QQ plots (quantile-quantile plots), for comparing two cumulative distribution functions.

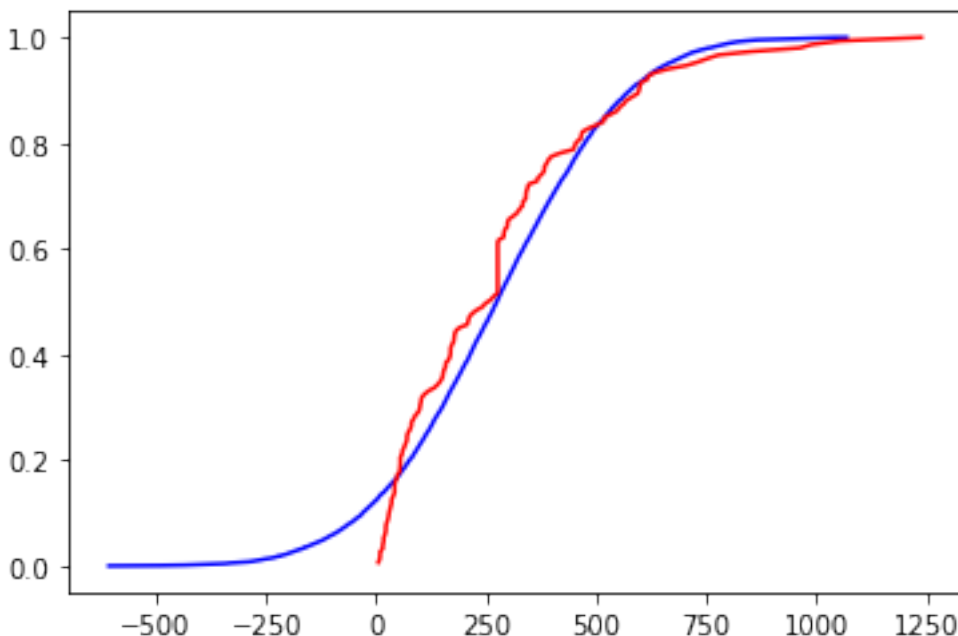
Here, we'll use what you saw in DataCamp.

```
import numpy as np

# create an ECDF from the data
ecdf_xs = sales_df['sales'].sort_values()
ecdf_ys = np.arange( 1, len(ecdf_xs)+1 ) / len(ecdf_xs)

# simulate a normal CDF with the same mean and std
sample_mean = ecdf_xs.mean()
sample_std = ecdf_xs.std()
samples = np.random.normal( sample_mean, sample_std, size=10000 )
normal_xs = np.sort( samples )
normal_ys = np.arange( 1, len(normal_xs)+1 ) / len(normal_xs)

# plot them on the same graph
plt.plot( normal_xs, normal_ys, 'b-' )
plt.plot( ecdf_xs, ecdf_ys, 'r-' )
plt.show()
```



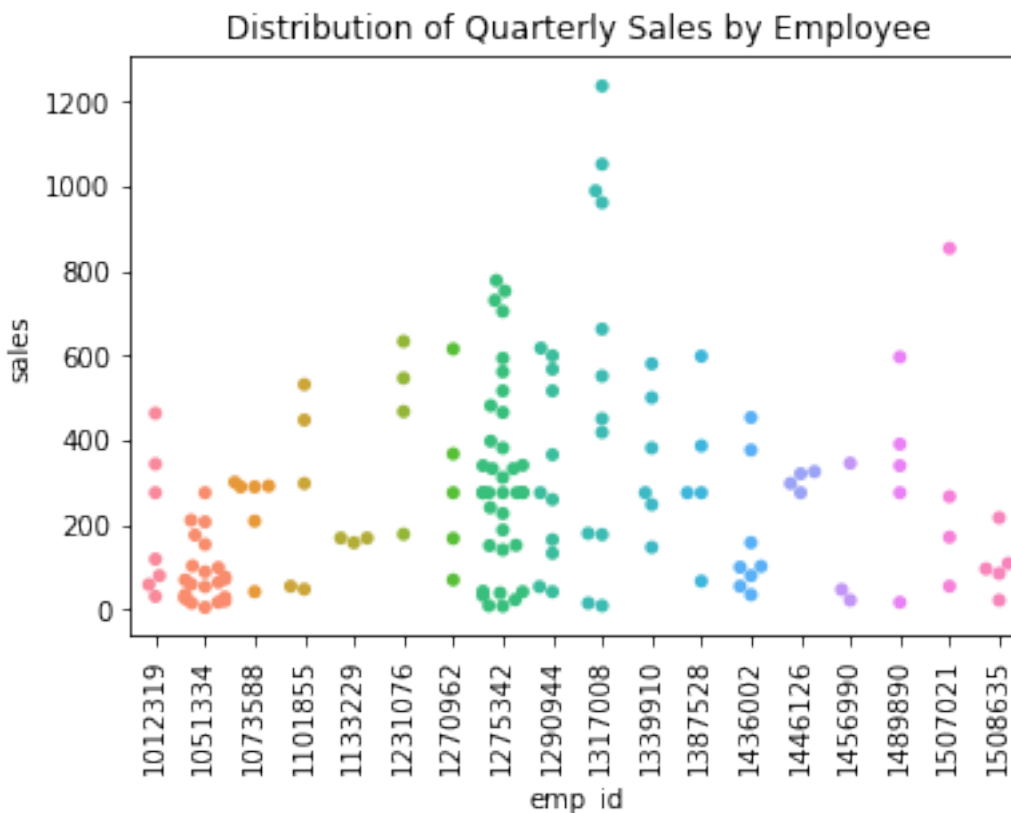
This case is hard to judge visually. The graphs are quite different for the leftmost 30% of the graph, and somewhat different for the middle, only converging at the end. If the project you're working on is something quick and dirty that just needs to be approximate, you might call this distribution close enough to normal. But if your project demands high accuracy, such as something in health care, you should resort to official statistical tests for normality of an empirical

distribution. We do not cover those in MA346.

1.10.6 What if I have lots of columns of data?

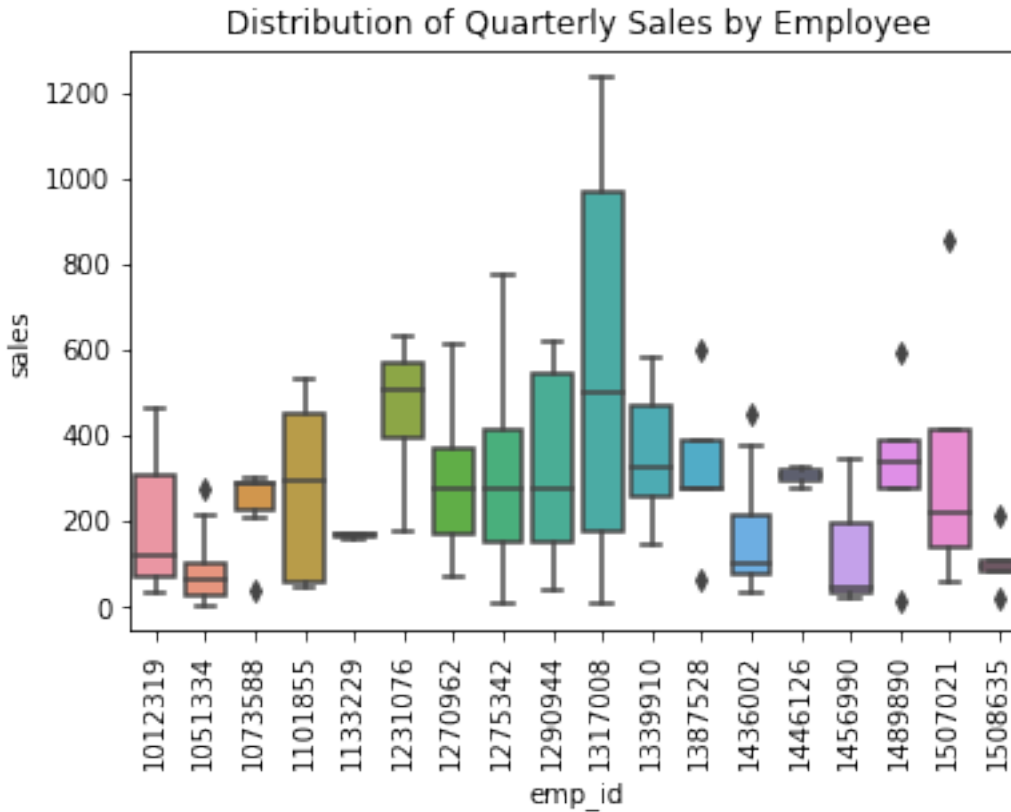
If you want to compare them as distributions, then all of the Seaborn plotting commands from the previous section still apply. They will show multiple distributions side-by-side, horizontally or vertically. Here are two examples.

```
sns.swarmplot( x='emp_id', y='sales', data=sales_df )
plt.title( 'Distribution of Quarterly Sales by Employee' )
plt.xticks( rotation=90 )
plt.show()
```



When showing only one variable (earlier), a box plot was quite boring. But when showing many variables, the simplicity of a box plot helps reduce visual clutter and make the variables much easier to compare.

```
sns.boxplot( x='emp_id', y='sales', data=sales_df )
plt.title( 'Distribution of Quarterly Sales by Employee' )
plt.xticks( rotation=90 )
plt.show()
```



What if we wanted to plot the four price distributions in the REGI dataset, the open, close, low, and high prices, side-by-side? Right now, these are stored in three separate columns in the data. But as you can see from the code above, Seaborn expects the data to be in a single column, and it will use a separate column to split the values into categories.

Of course, we know how to combine four columns of related data into one based on our work in a previous week—it's melting!

```
melted_df = regi_df.melt( id_vars=['Date'], value_vars=['Open', 'Close', 'Low', 'High'],
                          var_name='Type of price', value_name='Price' )
melted_df.head()
```

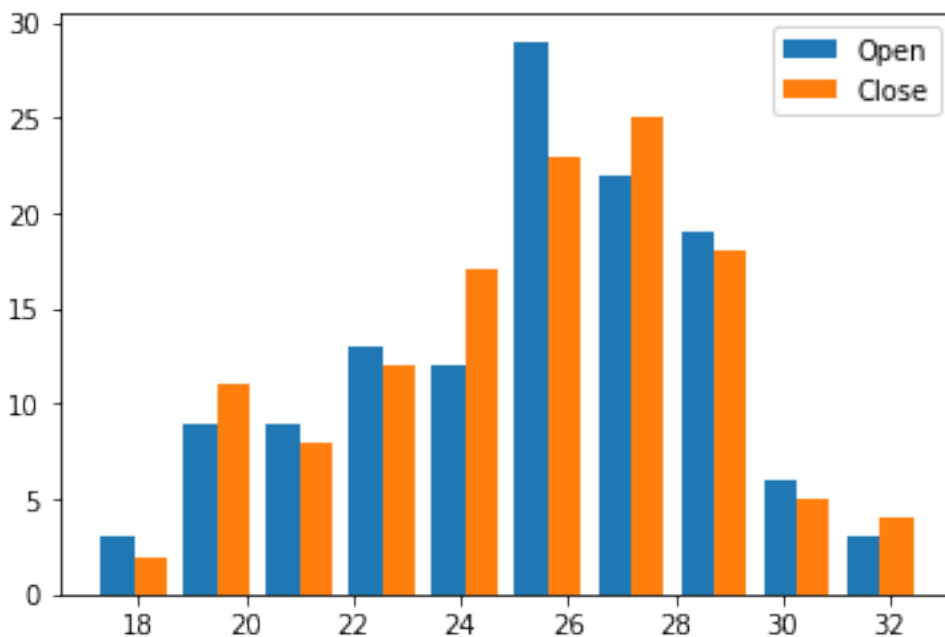
	Date	Type of price	Price
0	2-Jan-20	Open	27.21
1	3-Jan-20	Open	28.16
2	6-Jan-20	Open	28.53
3	7-Jan-20	Open	28.17
4	8-Jan-20	Open	26.37

```
sns.swarmplot( x='Type of price', y='Price', data=melted_df )
plt.title( 'Distribution of REGI Prices' )
plt.show()
```



And you can use the old, trusty histogram to compare distributions as well. Simply pass an array of Series instead of just one Series when calling `plt.hist()`.

```
plt.hist( [ regi_df['Open'], regi_df['Close'] ], label=[ 'Open', 'Close' ] )
plt.legend()
plt.show()
```

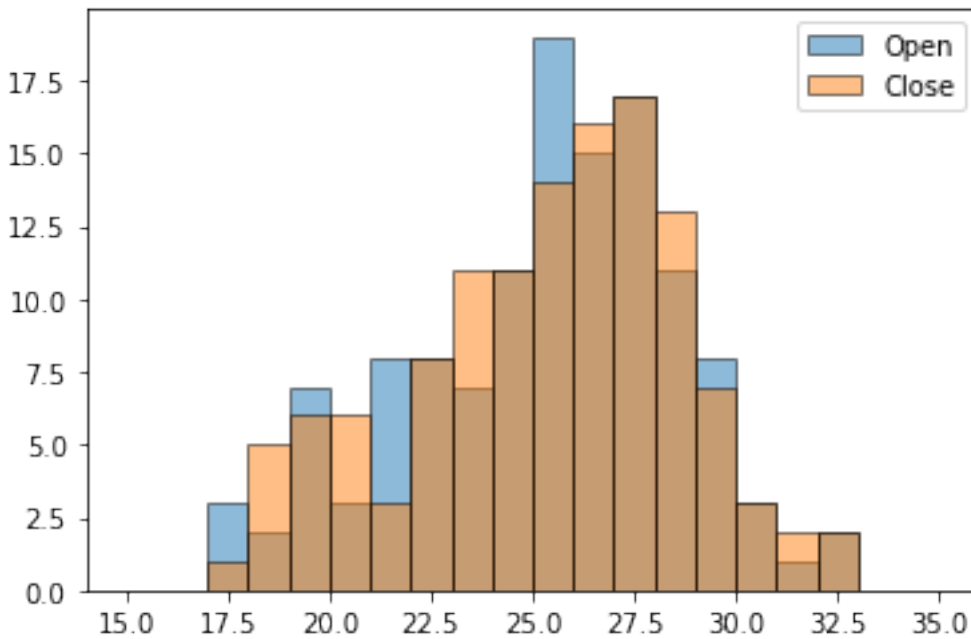


The REGI dataset is already set up for us to do this, because each distribution is in its own column. If it had not been so (but had been like the sales data, for instance), recall that the opposite of melting is pivoting, and that would get the data

in the needed form.

It's also possible to do overlapping histograms with transparent bars, but to get it to look good, you need to create the bin boundaries in advance and tell each histogram to use the same boundaries. Otherwise, `plt.hist()` will choose different bins for each Series of data.

```
bins = np.linspace( 15, 35, 21 ) # 20 bins from x=15 to x=35
plt.hist( regi_df['Open'], bins, label='Open', alpha=0.5, edgecolor='black' )
plt.hist( regi_df['Close'], bins, label='Close', alpha=0.5, edgecolor='black' )
plt.legend()
plt.show()
```

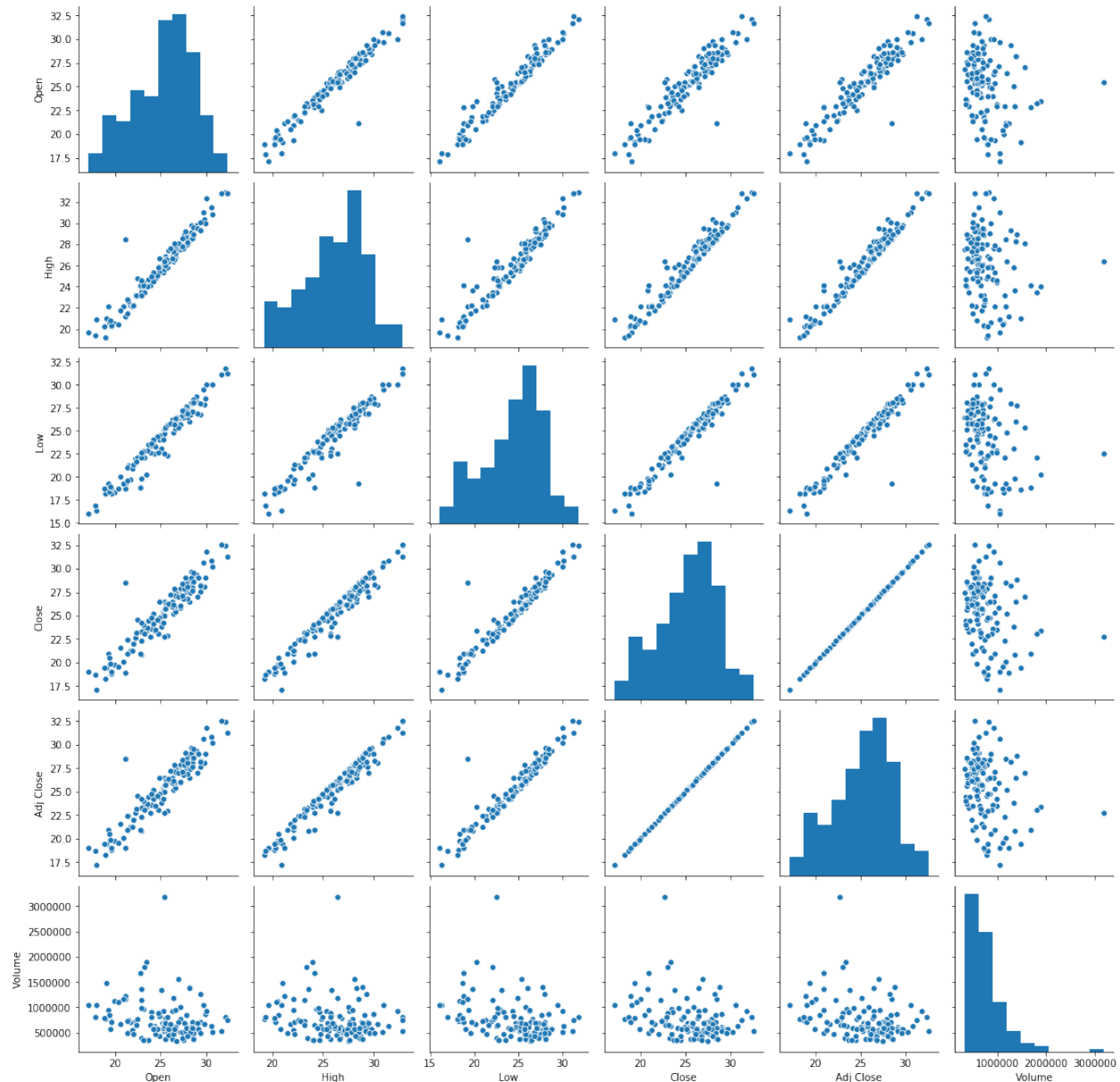


There's a lot more that could be said about plotting distributions; for instance, [here's a cool blog post](#) about how to make an even more beautiful plot that compares several distributions.

1.10.7 What if I need to know if the columns are related?

DataCamp showed you two visualizations for this. One focuses on giving you some visual intuition for whether the variables are related, by showing you the shape of all possible scatterplots of your data. It's called a pair plot because it pairs up the variables in every possible way. Let's try it on the REGI dataset; the explanation follows the picture.

```
sns.pairplot( regi_df )
plt.show()
```



The histograms shown along the diagonal of this graph are histograms of each variable, which are not the interesting part of the visualization.

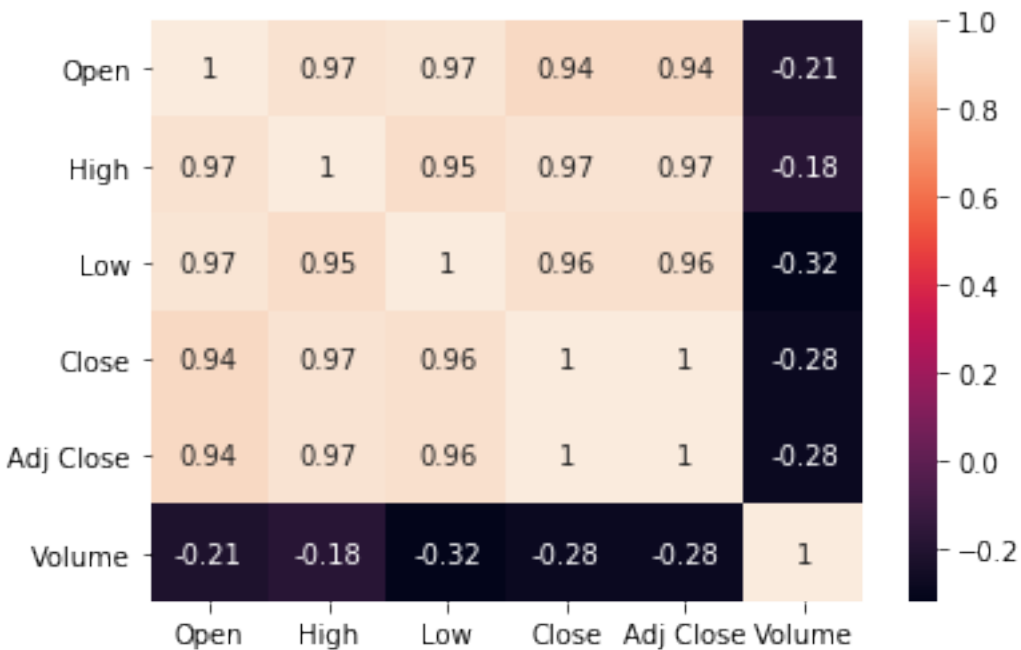
Next, take a look at the scatterplots that are *not* in the last row or last column. Almost all of them show a very tight linear relationship, but this is unsurprising because of the meaning of the data. For instance, the leftmost scatterplot in the second row relates the High price of a stock with the Open price of the stock on the same day. Because the stock opens and closes at approximately the same price on most days (no enormous fluctuations in any one day), these numbers are always close together, and thus highly correlated. The same goes for all the histograms except the final row and final column.

The final row and final column include the Volume variable. One might naturally wonder whether the volume of the stock traded on a day correlates to anything about the value of the stock on that day. In the case of Renewable Energy Group, Inc., for the first half of 2020, the answer seems to be no. There does not seem to be any discernable relationship in those histograms; they're just fuzzy blobs of data points.

Earlier I mentioned that `sns.pairplot()` was the technique that would give us some visual intuition for relationships,

and it did. But there is another visualization technique that doesn't show us as much visually, but gives us more easy-to-read measurements of the relationships among the variables. It's a heat map of the covariance matrix.

```
numeric_columns_only = regi_df.drop( 'Date', axis=1 )
correlation_coefficients = np.corrcoef( numeric_columns_only, rowvar=False )
sns.heatmap( correlation_coefficients, annot=True )
plt.xticks( np.arange(6)+0.5, numeric_columns_only.columns )
plt.yticks( np.arange(6)+0.5, numeric_columns_only.columns, rotation=0 )
plt.show()
```



Of course, because we used the same data, we still find out that all the prices are highly correlated (because they're organized by day) and the volume isn't really correlated much with anything. But it's much easier to tell both the correlations and the lacks of correlation when we have hard numbers to look at, rather than having to estimate it ourselves from shapes.

1.10.8 Summary of plotting tools

I know that was a huge amount to take in! So let's make it simpler:

With one numeric column of data:

If you want to see this	Then use this
Just the distribution's quartiles and outliers	Box plot
Simple approximation of the distribution	Histogram
Very good approximation of the distribution, maybe very wide	Swarm plot
Good approximation of the distribution, not too wide	Strip plot
Good approximation of a large distribution, smoothed	Violin plot
How similar is the distribution to normal?	Overlapping ECDFs

With two numeric columns of data:

If you want to see this	Then use this
A graph of my data, which is a function	Line plot
The shape of my data, which is a relation	Scatter plot
The shape of my data, which is a relation, plus each variable's distribution	Joint plot
The line of best fit through my data	<code>sns.lmplot</code>

With many numeric columns of data:

If you want to see this	Then use this
The quartiles and outliers of each	Side-by-side box plots
Simple approximation of the distributions	Histograms with side-by-side bars
Very good approximation of each distribution (can't fit too many)	Side-by-side swarm plots
Good approximation of each distribution (can fit more)	Side-by-side strip plots
Good approximation if the distributions are large (will be smoothed)	Side-by-side violin plots
The shape of all possible two-column relationships	Pair plot
A measurement of all possible correlations	Heat map of correlation coefficients

1.10.9 Techniques *not* to use (and why)

You may notice that we did not cover **pie charts** anywhere in this tutorial. Matplotlib can certainly produce pie charts for you, but visualization experts recommend against them, because viewers tend to have trouble assessing the exact meanings of the shapes. It's much harder to compare how much bigger one pie slice is to another than it is to compare, say, two bars on a histogram, or to points on a graph. So I suggest you avoid pie charts.

We also did not cover **bubble charts** anywhere in this tutorial. (A bubble chart is one in which each data point is plotted by a large circle, proportional to one of the variables in the data.) These are very popular in modern data visualization because they are eye-catching and attractive. But visualization experts recommend against these as well, because each person perceives the bubble sizes differently. For example, some people perceive the magnitude of a bubble on a graph in proportion to its radius, some perceive it in proportion to its area, and others are somewhere in between.

Visualization is a type of communication, and doing it well means focusing on the message you want to convey. Using a visualization that gives each viewer a different message is a bad idea. Unpredictability of viewer response is undesirable. So I suggest you avoid bubble charts as well.

We did not cover charts with **3D elements**, as Microsoft Excel often creates. This is because those elements also tend to distort the viewer's perception of the data and make it unclear exactly how extreme (or not) they're perceiving what you're showing. Thus we avoid any 3D elements in charts for the same reason we avoid bubble charts.

Finally, DataCamp showed you how to fit polynomial models to data using `sns.regplot()`. But I did not cover it here, because it is dangerous to dive into polynomial models without a solid grounding in mathematical modeling, which this course does not cover. Before using a polynomial model, you would need a solid, domain-specific reason to believe that such a model is applicable, or `sns.regplot()` will (obediently) produce results that are unreliable if used for prediction. Consequently, I won't cover `sns.regplot()` in MA346.

1.10.10 What about plot styles?

I didn't cover plot styles here, but there's nothing wrong with them. I simply left them out because most of them are only cosmetic; see [this week's section in the DataCamp cheat sheet](#) for details on items like `sns.set()`, `plt.subplot()`, and `plt.style`.

There are also some good blog posts on Matplotlib styles you might want to check out, such as [this](#) or [this](#).

But there is one stylistic element I want to highlight: DataCamp showed that `plt.annotate()` can be used to place text on a plot, which can be very useful for drawing a viewer's attention to the part of the graph that you want them to focus on. Consider the following graph, which we produced earlier, but now with a prominent annotation to explain why sales were so high one year.

```
plt.plot( grouped_df.index, grouped_df['sales'], '-o' ) # dots and lines
plt.title( 'Yearly Sales', fontdict={ "fontsize": 25 } )
plt.xlabel( 'Year' )
plt.ylabel( 'Total Sales' )
plt.ylim( [ 0, 10000 ] )
plt.annotate( 'Competitor\nflooded', xy=(2017.5,8000),
             color='red', size=15, ha='right' )
plt.show()
```



1.10.11 There's so much more!

Because visualization is a huge topic, I list several Learning On Your Own opportunities for extending your visualization knowledge and sharing it with the rest of the class.

Learning on Your Own - Plot with Less Code

In some cases, you can plot data directly from pandas without needing to use Matplotlib. [Investigate this blog post for details](#) and decide on the best format by which to report that information to the class.

Learning on Your Own - Geographical Plots

Drawing data on a map is extremely common and useful, but we don't have time to cover it in today's notes. [Here's a blog post about an easy way to do so in Python](#), but you don't need to feel bound to that one. There are many map toolkits for use in Python-based visualizations. Feel free to choose the one you like best and decide on the best format by which to report on it to the class. As an example, try showing how housing costs vary across the U.S. by plotting the property values in the mortgage dataset from Week 3 on a map.

Learning on Your Own - Tableau

One of the most famous tools for data visualization in industry is Tableau. Although coding in Python, R, etc., is always the most flexible option, tools like Tableau are far easier and faster when you don't need maximal flexibility. Take a Tableau tutorial and report to the class on its key features. Ensure you cover how to get a copy of Tableau, how to get data into it, and what it's best at.

Learning on Your Own - Visualization Design Principles

I've suggested a few concepts [up above](#) that can guide you towards effective visualizations and away from ineffective ones. But there is a lot to learn about visualization design principles that we can't cover here. Consider checking out [this blog post](#) or [this free online book](#) and choosing about five important concepts you learn that are relevant to our work in MA346. Find a good way to report them to the rest of the class, and be sure to include plenty of visual examples in your work of what to do and what not to do.

1.11 Processing the Rows of a DataFrame

See also the slides that summarize a portion of this content.

1.11.1 Goal

Back in the early days of programming, when I was a kid, we wrote code with stone tools.



And when we wanted to work with all the elements of an array, we had no choice but to write a loop.

```
shipments_received = [ 6, 9, 3, 4, 0, 0, 10, 4, 7, 6, 6, 0, 0, 13 ]

total = 0
for num_received in shipments_received:
    total += num_received

total
```

68

Most introductory programming courses teach loops, and for good reason; they show up a lot in programming! But there are a few reasons we'll try to avoid loops in data work whenever we can.

The lesser reason is **readability**. Loops are always at least two lines of code in Python; the one above is three because it has to initialize the `total` variable to zero. Many alternatives to loops can be done in just one line of code, which is more readable.

The more important reason is **speed**. Loops in Python are not very efficient, and this can be a serious problem. In the final project for MA346 in Spring 2020, many students came to my office hours with a loop that had been running for hours, and they didn't know if or when it would finish. There are *many* ways to speed loops up, sometimes by just altering the loop, but usually by replacing the loop with something else entirely.

In fact, that's the purpose of this chapter: *What can I do to improve a slow loop?*

The title of the chapter mentions DataFrames specifically, because in data work we're almost always processing a DataFrame row-by-row. But many of the techniques we'll cover apply to many different kinds of loops, with or without DataFrames.

An added benefit is that improving (or replacing) loops with something faster often means writing shorter or clearer code as well, achieving improvements in readability at the same time.

1.11.2 The `apply()` function

The most common use of a loop is when we need to do the same thing to each element of a sequence of values. Let's see an example.

Baseball example

In an earlier homework assignment, I provided a cleaned dataset of baseball players' salaries. Let's take a look at the original version of the dataset when I downloaded it [from the web](#), before it was cleaned.

```
import pandas as pd
df = pd.read_csv( '_static/baseball-salaries.csv' )
df.head()
```

	salary	name	total_value	pos	years	avg_annual	team
0	\$ 3,800,000	Darryl Strawberry	\$ 3,800,000	OF	1 (1991)	\$ 3,800,000	LAD
1	\$ 3,750,000	Kevin Mitchell	\$ 3,750,000	OF	1 (1991)	\$ 3,750,000	SF
2	\$ 3,750,000	Will Clark	\$ 3,750,000	1B	1 (1991)	\$ 3,750,000	SF
3	\$ 3,625,000	Mark Davis	\$ 3,625,000	P	1 (1991)	\$ 3,625,000	KC
4	\$ 3,600,000	Eric Davis	\$ 3,600,000	OF	1 (1991)	\$ 3,600,000	CIN

The “years” column looks particularly annoying. Why does it say “1 (1991)” instead of just 1991? Let's take a look at some other rows...

```
df.iloc[14440:14445, :]
```

	salary	name	total_value	pos	years	\
14440	\$ 100,000	Steve Monson	\$ 100,000	P	1	(1990)
14441	\$ 28,000,000	Alex Rodriguez	\$ 275,000,000	DH	10	(2008–17)
14442	\$ 200,000	Mike Colangelo	\$ 200,000	OF	1	(1999)
14443	\$ 200,000	Mike Jerzembek	\$ 200,000	P	1	(1999)
14444	\$ 21,680,727	Alex Rodriguez	\$ 21,680,727	3B	1	(2006)

	avg_annual	team
14440	\$ 100,000	MIL
14441	\$ 27,500,000	NYN
14442	\$ 200,000	LAA
14443	\$ 200,000	NYN
14444	\$ 21,680,727	NYN

Aha, some entries in the “years” column represent multiple years. We might naturally want to split that column up into three columns: number of years, first year, and last year. Each is a little project all on its own, but we just want to look at one example, so let's consider just the task of extracting the first year from the text. If we wrote a loop, it might go something like this.

Using a loop

```
first_years = [ ]
for text in df['years']:
    if text[1] == ' ': # one-digit number of years
        first_years.append( int( text[3:7] ) )
    else: # two-digit number of years
        first_years.append( int( text[4:8] ) )
df['first_year'] = first_years

df.iloc[[0,14441],:] # quick spot check of our work
```

	salary	name	total_value	pos	years	\
0	\$ 3,800,000	Darryl Strawberry	\$ 3,800,000	OF	1	(1991)
14441	\$ 28,000,000	Alex Rodriguez	\$ 275,000,000	DH	10	(2008-17)

	avg_annual	team	first_year
0	\$ 3,800,000	LAD	1991
14441	\$ 27,500,000	NYN	2008

A loop over a list of values is what pandas' `apply()` function was made for. You write `df['column'].apply(f)` to apply the function `f` to every entry in the chosen column. For example, we could simplify our work above as follows. The differences are noted in the comments.

Using `apply()`

```
# No need to start with an empty list.
def get_first_year ( text ): # Function name helps explain the code.
    if text[1] == ' ':
        return int( text[3:7] ) # Clearer and shorter than append().
    else:
        return int( text[4:8] ) # Clearer and shorter than append().
df['first_year'] = df['years'].apply( get_first_year )

df.iloc[[0,14441],:] # same check as before
```

	salary	name	total_value	pos	years	\
0	\$ 3,800,000	Darryl Strawberry	\$ 3,800,000	OF	1	(1991)
14441	\$ 28,000,000	Alex Rodriguez	\$ 275,000,000	DH	10	(2008-17)

	avg_annual	team	first_year
0	\$ 3,800,000	LAD	1991
14441	\$ 27,500,000	NYN	2008

If we're honest, the code didn't get *that* much simpler. But `apply()` is especially nice if the function we want to write is a function that already exists. Here's a silly example, but it illustrates the point.

```
df['name_length'] = df['name'].apply( len )
df.head()
```

	salary	name	total_value	pos	years	avg_annual	\
0	\$ 3,800,000	Darryl Strawberry	\$ 3,800,000	OF	1	(1991)	\$ 3,800,000
1	\$ 3,750,000	Kevin Mitchell	\$ 3,750,000	OF	1	(1991)	\$ 3,750,000
2	\$ 3,750,000	Will Clark	\$ 3,750,000	1B	1	(1991)	\$ 3,750,000

(continues on next page)

(continued from previous page)

3	\$ 3,625,000	Mark Davis	\$ 3,625,000	P	1	(1991)	\$ 3,625,000
4	\$ 3,600,000	Eric Davis	\$ 3,600,000	OF	1	(1991)	\$ 3,600,000
	team	first_year	name_length				
0	LAD	1991	17				
1	SF	1991	14				
2	SF	1991	10				
3	KC	1991	10				
4	CIN	1991	10				

Using `apply()` will run a little faster than writing your own loop, but unless the `DataFrame` is really huge, you probably won't notice the difference, so speed is not a significant concern here. But switching to the `apply()` form sets us up nicely for a later speed improvement *we'll discuss further below*.

Although it's less often useful, you can use `df.apply(f)` to run `f` on each column of the `DataFrame`, or `df.apply(f, axis=1)` to run `f` on each row of the `DataFrame`.

There is, unfortunately, a related function `map()`. It behaves very similarly to `apply()`, with a few subtle differences. This is unfortunate because in computer programming more broadly, the concepts of “map” and “apply” are often used synonymously/interchangeably. So to have them behave almost the same (but slightly differently!) in pandas is unfortunate. Oh well. Here are the differences:

Feature	<code>apply()</code>	<code>map()</code>
You can use it on <code>DataFrames</code> , as in <code>df.apply(f)</code>	Yes	No
You can provide extra <code>args</code> or <code>kwargs</code>	Yes	No
You can use a dictionary instead of <code>f</code>	No	Yes
You can ask it to skip <code>NaNs</code>	No	Yes

Big Picture

In most programming contexts, including data work, if someone speaks of “mapping” or “applying” a function, they mean the same thing: Automatically running the function on each element of a list or series. The function for this is often called `map()` or `apply()`, as in pandas, but not always. In mathematics, it's called using a function “elementwise,” meaning on each element of a structure separately. In the popular language Julia, it's called “broadcasting” a function over an array or table.

The function that you give to `apply()` can't be just any function. Its input type needs to match the data type of the individual elements in the `Series` or `DataFrame` you're applying it to. Its output type will determine what kind of output you get. For example, the `get_first_year()` function defined above takes strings as input and gives integers as output. So using `apply(get_first_year)` will need to be done on a `Series` containing strings, and will produce a `Series` containing integers.

If you have a function that takes multiple inputs, you might want to bind some of the arguments so that it becomes a unary function and can be used in `apply()`. Or you can use the `args` or `kwargs` feature of `apply()`, but we won't cover that in these course notes. You can see a small example in [the pandas documentation](#). We will, however, take a look at the possibility of using a dictionary with `map()`, because it is extremely useful. We will consider a simple example application, but do a more sophisticated one in class.

Using map ()

Let's assume that the analysis we wanted to do cared only about whether the baseball player had an infield position (IF), outfield position (OF), was a pitcher (P), or a designated hitter (DH), and we didn't care about any other details of the position (such as first base vs. second base, or starting pitcher vs. relief pitcher). We'd therefore like to simplify the "pos" column and convert all infield positions to IF, and so on. First, let's see what all the positions are.

```
df['pos'].unique()
```

```
array(['OF', '1B', 'P', 'DH', '3B', '2B', 'C', 'SS', 'RF', 'SP', 'LF',
       'CF', 'RP'], dtype=object)
```

We could convert them with a big if statement, like you see here, but this is tedious and repetitive code.

```
def simpler_position ( pos ):    # BAD STYLE.  See better version below.
    if pos == 'P': return 'P'
    if pos == 'SP': return 'P'
    if pos == 'RP': return 'P'
    if pos == 'C': return 'IF'
    if pos == '1B': return 'IF'
    if pos == '2B': return 'IF'
    if pos == '3B': return 'IF'
    if pos == 'SS': return 'IF'
    if pos == 'OF': return 'OF'
    if pos == 'LF': return 'OF'
    if pos == 'CF': return 'OF'
    if pos == 'RF': return 'OF'
    if pos == 'DH': return 'DH'
```

```
df['simple_pos'] = df['pos'].apply( simpler_position )
df.head()
```

	salary	name	total_value	pos	years	avg_annual	\
0	\$ 3,800,000	Darryl Strawberry	\$ 3,800,000	OF	1 (1991)	\$ 3,800,000	
1	\$ 3,750,000	Kevin Mitchell	\$ 3,750,000	OF	1 (1991)	\$ 3,750,000	
2	\$ 3,750,000	Will Clark	\$ 3,750,000	1B	1 (1991)	\$ 3,750,000	
3	\$ 3,625,000	Mark Davis	\$ 3,625,000	P	1 (1991)	\$ 3,625,000	
4	\$ 3,600,000	Eric Davis	\$ 3,600,000	OF	1 (1991)	\$ 3,600,000	

	team	first_year	name_length	simple_pos
0	LAD	1991	17	OF
1	SF	1991	14	OF
2	SF	1991	10	IF
3	KC	1991	10	P
4	CIN	1991	10	OF

All the repetitive code is just establishing a simple relationship among some very short strings. We could store that same relationship in a dictionary with many fewer lines of code. Note that we must use `map()`, because `apply()` doesn't accept dictionaries.

```
df['simple_pos'] = df['pos'].map( {
    'P': 'P',    'SP': 'P',    'RP': 'P',    'C': 'IF',
    '1B': 'IF',  '2B': 'IF',  '3B': 'IF',  'SS': 'IF',
    'OF': 'OF',  'LF': 'OF',  'CF': 'OF',  'RF': 'OF',  'DH': 'DH'
} )
df.head()
```

	salary	name	total_value	pos	years	avg_annual	\
0	\$ 3,800,000	Darryl Strawberry	\$ 3,800,000	OF	1 (1991)	\$ 3,800,000	
1	\$ 3,750,000	Kevin Mitchell	\$ 3,750,000	OF	1 (1991)	\$ 3,750,000	
2	\$ 3,750,000	Will Clark	\$ 3,750,000	1B	1 (1991)	\$ 3,750,000	
3	\$ 3,625,000	Mark Davis	\$ 3,625,000	P	1 (1991)	\$ 3,625,000	
4	\$ 3,600,000	Eric Davis	\$ 3,600,000	OF	1 (1991)	\$ 3,600,000	

	team	first_year	name_length	simple_pos
0	LAD	1991	17	OF
1	SF	1991	14	OF
2	SF	1991	10	IF
3	KC	1991	10	P
4	CIN	1991	10	OF

In class, we will do a more complex example of applying a dictionary using `map()`. Before class, you may want to glance back at Exercise 3 from [the Chapter 2 notes](#), which shows you how to take two columns of a DataFrame representing a mathematical function and convert them into a dictionary for use in situations just like this one. And be sure to complete the homework about the NPR dataset before class as well, because we will use that in our example!

Parallel `apply()`

I mentioned earlier that converting a loop into an `apply()` or `map()` call doesn't gain us much speed. But it does make it easy for us to add a nice speed improvement. There's a Python package called `swifter` that you can install using the instructions on that page. Once it's installed, you can convert any code like `df['column'].apply(f)` easily into a faster version by replacing it with `df['column'].swifter.apply(f)`. That's all!

Under the hood, `swifter` is trying a variety of speedup mechanisms (many of which we discuss in this chapter) and deciding which of them works best for your situation. The most common one for large dataset is probably parallel processing. This means that if your computer has more than one processor core (which most modern laptops do), then it can process more than one entry of the data at once, each on a separate core.

Without `swifter`, you could accomplish the same thing with code like the following. (In fact, if you have trouble installing `swifter`, you can use this code instead.)

```
# Use Python's built-in multiprocessing module to find your number of cores.
import multiprocessing as mp
n_cores = mp.cpu_count()

# Create a "pool" of functions that can work at the same time and run them.
pool = mp.Pool( n_cores )
df['simple_pos'] = pool.map( simpler_position, df['pos'], n_cores )

# Clean up afterwards.
pool.close()
pool.join()

# See result.
df.head()
```

	salary	name	total_value	pos	years	avg_annual	\
0	\$ 3,800,000	Darryl Strawberry	\$ 3,800,000	OF	1 (1991)	\$ 3,800,000	
1	\$ 3,750,000	Kevin Mitchell	\$ 3,750,000	OF	1 (1991)	\$ 3,750,000	
2	\$ 3,750,000	Will Clark	\$ 3,750,000	1B	1 (1991)	\$ 3,750,000	
3	\$ 3,625,000	Mark Davis	\$ 3,625,000	P	1 (1991)	\$ 3,625,000	
4	\$ 3,600,000	Eric Davis	\$ 3,600,000	OF	1 (1991)	\$ 3,600,000	

(continues on next page)

(continued from previous page)

	team	first_year	name_length	simple_pos
0	LAD	1991	17	OF
1	SF	1991	14	OF
2	SF	1991	10	IF
3	KC	1991	10	P
4	CIN	1991	10	OF

1.11.3 Map-Reduce

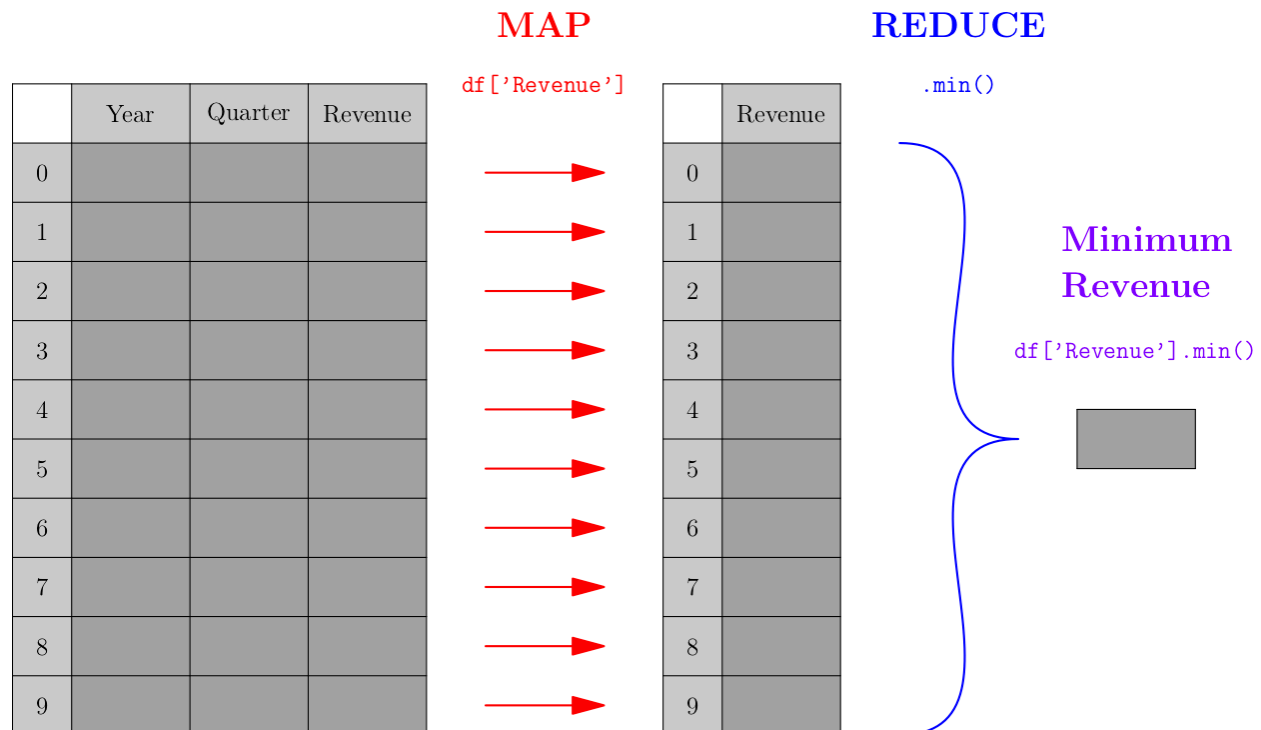
Big Picture

Both *map-reduce* and *split-apply-combine* are data manipulation buzzwords that you'll want to be familiar with, for

- thinking about your own data manipulation work,
- discussing that work with coworkers, and
- knowing what people are saying in, e.g., interviews.

This section covers map-reduce and the next section covers split-apply-combine.

A map-reduce process is one that takes any list, *maps* a specific function across all entries of the list, then *reduces* those outputs down to a single, smaller result. Consider the following picture, which shows a very simple map-reduce operation that takes a DataFrame about historic revenue numbers and computes the lowest revenue across all quarters.



Let's actually do the above computation on some small sample (fictional) data:

```
# setup - example tiny dataset
rev_quarters = pd.DataFrame( {
```

(continues on next page)

(continued from previous page)

```
'Year'      : [ 2010, 2010, 2010, 2010, 2011, 2011, 2011, 2011, 2012, 2012 ],
'Quarter'   : [    1,    2,    3,    4,    1,    2,    3,    4,    1,    2 ],
'Revenue'   : [ 177, 186, 167, 263, 180, 193, 189, 281, 201, 210 ]
} )
rev_quarters
```

	Year	Quarter	Revenue
0	2010	1	177
1	2010	2	186
2	2010	3	167
3	2010	4	263
4	2011	1	180
5	2011	2	193
6	2011	3	189
7	2011	4	281
8	2012	1	201
9	2012	2	210

```
# map-reduce work, one line:
rev_quarters['Revenue'].min()
```

167

As mentioned earlier, “map” is a synonym for “apply,” so the first step of the process applies the same operation to all rows of the DataFrame; in this case, that operation extracts the revenue from the row. The “reduce” operation in this case is a simple `min()` operation, but it can be something more complex.

So a map-reduce operation involves two functions, the first performing a `map()` operation (as discussed earlier), and the second doing something new. The function used for the reducing step must be something that takes an entire list or series as input and produces a single value as output. The `min()` operation was used in the example above, but other operations are common, such as `max()`, `sum()`, `len()`, `mean()`, `median()`, and more.

Argmin and argmax

A very common function that shows up in statistics is called `argmin` (and its companion `argmax`). These are also implemented in pandas and are very useful in map-reduce situations. In the example above, let’s say we didn’t want to know the minimum revenue, but we wanted to know in which quarter the minimum revenue happened. We can replace `min` in the above code with `argmin` to ask that question.

```
rev_quarters['Revenue'].argmin()
```

2

The `argmin` function is short for “the argument that yields the minimum,” or in other words, what value would I need to supply as *input* to the map function to get the minimum output? In this case, the map function takes each row and extracts its revenue, so we’re asking pandas, “When you found the minimum revenue, which row was the input?” The answer was row 2, and we can see that it’s the correct row as follows.

```
rev_quarters.iloc[2]
```

Year	2010
Quarter	3

(continues on next page)

(continued from previous page)

```
Revenue      167
Name: 2, dtype: int64
```

While the pandas documentation for `argmin` and `argmax` suggest that they return multiple values in the case of ties, this doesn't seem to be true. They seem to return the first index only. You can therefore always rely on the result of `argmin/argmax` being a single value, never a list or series. If you want the indices of all max/min entries, you will need to compute it another way.

Map-reduce example: sample standard deviation

The formula for the standard deviation of a sample of data should be familiar to you from GB213.

$$s = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1}}$$

Let's assume we've already computed the mean value \bar{x} . Then computing the standard deviation is actually a map-reduce operation. The map function takes each x_i as input and computes $(x_i - \bar{x})^2$ as output. The reduce operation then does a sum, divides by $n - 1$, and takes a square root. We could code it like so:

```
import numpy as np

example_data = df['first_year']
x_bar = example_data.mean()

def map_func ( x ):
    return ( x - x_bar ) ** 2
def reduce_func ( data ):
    return np.sqrt( data.sum() / ( len(data) - 1 ) )

reduce_func( example_data.map( map_func ) )
```

```
7.926156939014573
```

Of course, we didn't have to code that. There's already an existing standard deviation function built into pandas, and it gives almost exactly the same answer. (I suspect theirs does something more careful with tiny issues of accuracy than my simple example does.)

```
example_data.std()
```

```
7.926156939014146
```

But it is still important to notice that the pattern in computing a sample standard deviation is a map-reduce pattern, because we cannot always rely on pandas to do computations for us. For instance, if the data we were dealing with were many gigabytes spread over a database, we couldn't load it all into a pandas DataFrame in memory and then call `data.std()` to get our answer.

There are specialized tools in the industry for applying the map-reduce paradigm to databases (even if the database is enormous and spread over many different servers). One famous example is [Apache Spark](#), but there are many.

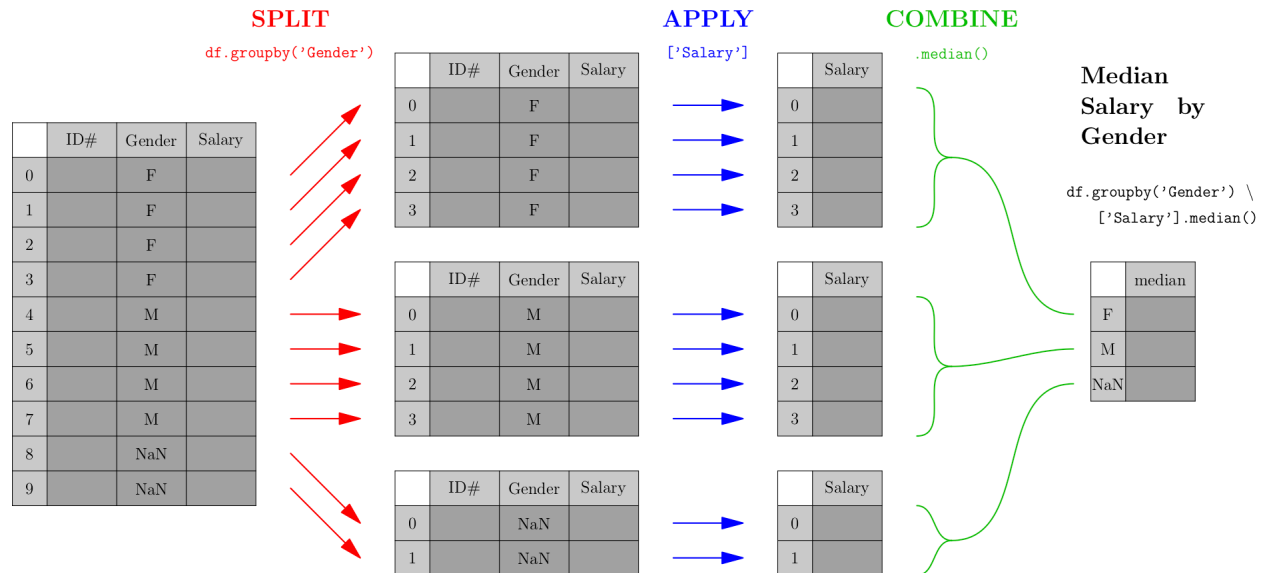
Many more examples of map-reduce from math and statistics could have been shown instead of the one above. Any time a list of values collapses to give a single result, map-reduce is behind it. This happens for summations, approximations of integrals (e.g., trapezoidal rule), expected values, matrix multiplication, computing probabilities from trees of possible outcomes, any weighted averages (chemical concentrations, portfolio values, etc.), and many more.

1.11.4 Split-Apply-Combine

Data scientist and R developer Hadley Wickham seems to coin lots of important phrases. Recall from *the Chapter 5 notes* that he introduced the phrase “tidy data.” He also introduced the phrase “split, apply, combine,” in [this paper](#).

It is another extremely common operation done on DataFrames, and it is closely related to map-reduce, as we will see below.

Let’s say you were concerned about pay equity, and wanted to compute the median salary across your organization, by gender, to get a sense of whether there were any important discrepancies. The computation would look something like the following. (We assume that the gender column contains either M for male, F for female, or a missing value for those who do not wish to classify.)



As you can see from the picture, the first phase (called “split”) breaks the data into groups by the categorical variable we care about—in this case, gender. After that, each smaller DataFrame undergoes a map-reduce process, and the results of each small map-reduce get aggregated into a result, indexed by the original categorical variable.

Note that the output type of the split operation (which, in pandas, is a `df.groupby()` call) is *NOT* a DataFrame, but rather a collection of DataFrames. It is essential to follow a `df.groupby()` call with the apply and combine steps of the process, so that the result is a familiar and usable type of object again—a pandas DataFrame.

The easiest type of split-apply-combine is shown in the picture above and can be done with a single line of code. We’ll compute minimum revenue by year with the DataFrame from our map-reduce example.

```
rev_quarters.groupby('Year')['Revenue'].min()
```

```
Year
2010    167
2011    180
2012    201
Name: Revenue, dtype: int64
```

Split-apply-combine is actually a specific type of pivot table. Thus split-apply-combine operations can be done on data in Excel as well, using its pivot table features. We can even use `df.pivot_table()` to mimic the above procedure, as follows. (Because we don’t need data separated into separate columns, we don’t provide a columns variable.)

```
rev_quarters.pivot_table(index=['Year'], columns=[], values='Revenue', aggfunc='min')
↪
```

	Revenue
Year	
2010	167
2011	180
2012	201

1.11.5 More on math in Python

Arithmetic in formulas

Recall that pandas is built on NumPy, and in [Chapter 9 of the notes](#) we talked about NumPy's support for vectorization. If we have a Series `height` containing heights in inches and we need instead to have it in centimeters, we don't need to do `height.apply()` and give it a conversion function, because we can just do `height * 2.54`. NumPy automatically *vectorizes* this operation, spreading the "times 2.54" over each entry in the `height` array.

This is quite natural, because we have mathematical notation that does the same thing (in math, not Python). If you've taken a class involving vectors, you know that vector addition $\vec{x} + \vec{y}$ means to do exactly what NumPy does—add the corresponding entries in each vector. Similarly, scalar multiplication $s\vec{x}$ means to multiply s by each entry in the vector \vec{x} , just like `height * 2.54` does in Python. So NumPy is not inventing something strange here; it's normal mathematical stuff.

All the basic mathematical operations are built into NumPy. For example, if we have created a linear model $\hat{y} = \beta_0 + \beta_1 x$ with parameters stored in Python variables β_0 and β_1 , we can apply it to an entire series of inputs `xs` at once with the following code, because NumPy knows how to spread both `+` and `*` across arrays.

```
y_hat = beta_0 + beta_1 * xs
```

In fact, if we had actual `ys` that went with the `xs`, we could then compute a list of residuals all at once with `y_hat - ys`, or even compute the RMSE (root mean squared error) with code like this.

```
np.sqrt( np.sum( ( y_hat - ys ) ** 2 ) / len( ys ) )
```

The subtraction with `-` and the squaring with `** 2` would all be spread across arrays of inputs correctly, because NumPy comes with code to support doing so.

Conditionals with `np.where()`

This removes a lot of the need for both loops and `apply()/map()` calls, but not all. One of the first things that makes us think we might need a loop is when a conditional computation needs to be done. For instance, let's say we were given a dataset like the following (made up) example.

```
patients = pd.DataFrame( {
    'id' : [ 100615, 51, 100616, 83, 100607, 100618, 19, 65 ],
    'height' : [ 72, 158, 75, 173, 68, 67, 163, 178 ],
    'dose' : [ 2, 0, 2.5, 2, 0, 2, 2.5, 0 ]
} )
patients
```

	id	height	dose
0	100615	72	2.0
1	51	158	0.0
2	100616	75	2.5
3	83	173	2.0

(continues on next page)

(continued from previous page)

4	100607	68	0.0
5	100618	67	2.0
6	19	163	2.5
7	65	178	0.0

Let's imagine that we then found out that it was the result of merging data from two different studies, one done in the U.S. and one done in France. The data with IDs that begin with 100 are from the U.S. study, where heights were measured in inches. The data with two-digit IDs are from the French study, where heights were measured in cm. We need to standardize the units.

We can't simply convert to cm with `patients['height'] * 2.54` because that would apply the conversion to all data rather than just the measurements in inches. We need some conditional logic, perhaps using an `if` statement, to be selective. Our first inclination might be a loop.

```
# before changing the contents, make a backup, for use later.
backup = patients.copy()

# solving the problem with a loop:
for index, row in patients.iterrows():
    if row['id'] > 100000: # US data
        patients.loc[index, 'height'] *= 2.54
patients
```

	id	height	dose
0	100615	182.88	2.0
1	51	158.00	0.0
2	100616	190.50	2.5
3	83	173.00	2.0
4	100607	172.72	0.0
5	100618	170.18	2.0
6	19	163.00	2.5
7	65	178.00	0.0

Note that `row['height'] *= 2.54` actually wouldn't alter the DataFrame, so we're forced to use `patients.loc[]` instead.

But if you were trying to follow the advice in this chapter of the notes, you might switch to an `apply()` function instead. The trouble is, it's a bit annoying to do, because we need the `if` to operate on the "id" column and the conversion to operate on the "height" column, so which one do we call `apply()` on? We can call `apply()` on the whole DataFrame, but the loop is actually simpler in that case!

The solution here is to use NumPy's `np.where()` function. It lets you select just which rows should get which type of computation, like so:

```
# restore the original data:
patients = backup.copy()

# solution with np.where():
patients['height'] = np.where( patients['id'] > 100000, patients['height'] * 2.54,
↪ patients['height'] )
patients
```

	id	height	dose
0	100615	182.88	2.0
1	51	158.00	0.0
2	100616	190.50	2.5

(continues on next page)

(continued from previous page)

3	83	173.00	2.0
4	100607	172.72	0.0
5	100618	170.18	2.0
6	19	163.00	2.5
7	65	178.00	0.0

The `np.where()` function works just like `=IF()` does in Excel, taking three inputs, a conditional, an “if” result, and an “else” result. But the difference is that `np.where()` is vectorized, effectively doing an Excel `=IF()` on each entry in the Series separately. You can read an `np.where()` function just like a sentence:

Where patient id is over 100000, do patient height times 2.54, otherwise just keep the original height.

In summary, thanks to `np.where()`, even many conditional computations don’t require a loop or an apply; they can be done with NumPy vectorization as well.

Speeding up mathematics

There are also some very impressive tools for speeding up mathematical operations in NumPy a *LOT*. I will not cover them here, but will list each of the following as an opportunity for Learning On Your Own. Note that these are relevant only if you have a very large dataset over which you need to do complex mathematical computations, so that you notice pandas behaving slowly, and thus you need a speed boost.

Learning on Your Own - CuPy (fastest option)

Doing certain types of computations can be sped up significantly by using graphics cards (originally designed for gaming rather than data science) instead of the computer’s CPU (which does all the non-graphics computations). See [this blog post](#) for information on CuPy, a Python library for harnessing your GPU to do fast arithmetic.

CuPy requires you to first describe to it the computation you’ll want to do quickly, and it will compile it into GPU-friendly code that you can then use. This is an extra level of annoyance for the programmer, but often produces the fastest results.

Learning on Your Own - NumExpr (easiest option)

If you’ve already got some code that does the arithmetic operation you want on NumPy arrays (or pandas Series, which are also NumPy arrays), then it’s pretty easy to convert that code to use NumExpr. It doesn’t give as big a speedup as CuPy, but it’s easier to set up. See [this blog post](#) for details, and note the connection to `pd.eval()`.

Learning on Your Own - Cython (most flexible)

The previous two options work only for speeding up arithmetic. To speed up any operation (including string manipulation, working with dictionaries, sets, or any Python class), you’ll need Cython. This is a tool for converting Python code into C code automatically, without your having to learn to program in C. C code almost always runs significantly faster than Python code, but C is much less easy to use, especially for data work. See [this tutorial](#) on using Cython in Jupyter, plus the example below.

Let’s say I have the following function that computes $n!$, the product of all positive integers up to n . (This is not the best way to write this function, but it’s just an example.)

```
def factorial ( n ):
    result = 1
    for i in range( 1, n+1 ):
```

(continues on next page)

(continued from previous page)

```
        result *= i
    return result

factorial( 5 )
```

```
120
```

I can ask Jupyter to compile this into C code for me, so that it runs faster, as follows.

First, use one cell of the notebook to load the Cython extension.

```
%load_ext cython
```

Then, ask Cython to convert your Python code into C. This requires giving it some hints (highlighted in the comments below) about the data types of the variables. In this simple case, they're all integers.

```
%%cython -a
def factorial ( int n ):      # n is an integer
    cdef int result, i        # so are result and i
    result = 1
    for i in range( 1, n+1 ):
        result *= i
    return result
```

If you run the above code in Jupyter, it will show you an interactive display of the code it created and how much speedup you can expect. The function still generates the same outputs as before, but typically much faster. How much faster? Check out the tutorial linked to above for more information.

1.11.6 So do we *always* avoid loops?

No, there are some times when you might still want to avoid loops.

When to opt for a loop

The two most prominent times to choose loops are these.

1. If the code you're running is a search for one thing, and you want to stop once it's found, a loop might be best. Take the home mortgage database of 15 million records, for example. Let's say you were looking for an example of a Hispanic male in Nevada applying for a mortgage for a rental property. If you ask pandas to filter the dataset, it will examine all 15M rows and give you *all* the ones fitting these criteria. But you just needed one. Maybe you'd find it in the first 50,000 rows and not need to search the other 14.95 million! A loop definitely has the potential to be faster in such a case.
2. Sometimes the computation you're doing involves comparing one row to adjacent rows. For example, you might want to find those days when the price of a stock was significantly more or less than it was on the two adjacent days (one before and one after). Although it's possible to do this without a loop, the code is a harder to write and to read, as you can see in the example below. With a loop, it's not as fast, but it's clearer. So if speed isn't an issue, use the loop.

Let's see how we might write the code for the stock example just given, but instead of stock data, we'll use the (made up) quarterly revenue data from earlier.

```
# get just the column I care about:
revenues = rev_quarters['Revenue']

results = [ ]
# For each quarter except the first and last...
for index in revenues.index[1:-1]:
    # If it's bigger than the previous and the next...
    if revenues.loc[index] > revenues.loc[index-1] and \
        revenues.loc[index] > revenues.loc[index+1]:
        results.append( index ) # Save it for later

# Show me just the quarters I saved.
rev_quarters.iloc[results,:]
```

	Year	Quarter	Revenue
1	2010	2	186
3	2010	4	263
5	2011	2	193
7	2011	4	281

Compare that to the same results computed using vectorization in NumPy rather than a loop. If the data were large, this implementation would be faster, but it's definitely not as clear to read.

```
# Get all but first and last, for searching.
to_search = rev_quarters.iloc[1:-1]

# Compute arrays of previous/next quarters, for comparison.
previous_rev = rev_quarters.iloc[:-2]
next_rev = rev_quarters.iloc[2:]

# Adjust indices so they match the to_search Series.
previous_rev.index = previous_rev.index + 1
next_rev.index = next_rev.index - 1

# Do the computation using NumPy vectorized comparisons.
to_search[( to_search['Revenue'] > previous_rev['Revenue'] ) \
          & ( to_search['Revenue'] > next_rev['Revenue'] )]
```

	Year	Quarter	Revenue
1	2010	2	186
3	2010	4	263
5	2011	2	193
7	2011	4	281

Any time when speed isn't an issue, and you think the clearest way to write the code is a loop, then go right ahead and write clear code! Loops aren't always bad.

Factoring computations out of the loop

Sometimes what's making a loop slow is a repeated computation that doesn't need to happen inside the loop. The *loop variable* is the variable that immediately follows the `for` statement in a loop. In the loop example above, that's the `index` variable. If there were any computation inside the loop that didn't use the `index` variable, we could bring that computation outside the loop, doing it once, before the loop, and saving time.

For example, in the final project some students did for MA346 in Spring 2020, some teams had a loop that processed a large database of baseball players, and tried to look their names up in a different database. It went something like this:

```
for name in baseball_df['player name']:
    if name in other_df["Player's Name"]:
        # then do stuff here
```

Because the two DataFrames were very large, this loop took literally hours to run on students' laptops, and made it impossible for them to improve their code in time to finish the project. The first thing I suggested was to change the code as follows.

```
for name in baseball_df['player name']:
    if name in other_df["Player's Name"].unique():
        # then do stuff here
```

The `.unique()` function computes a smaller list from `other_df['name']`, in which each name shows up only once. This meant a smaller search to do, and sped up the loop, but even so, it wasn't fast enough. It still took about 30 minutes, which made it hard for students to iteratively improve their code.

But notice that the loop variable, `name`, doesn't appear anywhere in the computation of `other_df["Player's Name"].unique()`. So we're asking Python to compute that list of unique names over and over, each time through the loop. Let's bring that outside the loop so we have to do it only once.

```
unique_name_list = other_df["Player's Name"].unique()
for name in baseball_df['player name']:
    if name in unique_name_list:
        # then do stuff here
```

This loop ran much faster, and most students were able to use it to do the work of their final project.

Note that this advice, factoring out a computation that does not depend on the loop variable, is sort of the opposite of abstraction. In abstraction, you make the list of all the variables that your computation *does* depend on, and move those up to the top, as input parameters. Here we're taking a look at which variables our computation *doesn't* depend on, so that we can move the computation itself up to the top, so it is done outside the loop.

Knowing how long you'll have to wait

Few things are more frustrating than running a code cell and seeing the computer just sit there doing nothing. We start to wonder whether it will take 15 seconds to process the data, and we should just have a little patience, or 15 minutes and we should go get a coffee, or 15 hours and we should give up and rewrite the code. Which is it? How can we tell except just waiting?

There are two easy ways to get some feedback as your loop is progressing. The easiest one is to install the `tqdm` module, whose purpose is to help you see a progress bar for a long-running loop. After following [tqdm's installation instructions](#) (using `pip` or `conda`), just import the module, then take the Series or list over which you're looping and wrap it in `tqdm(...)`, as in the example below.

```
from tqdm.notebook import tqdm
```

(continues on next page)


(continued from previous page)

```

results = [ ]
for index in tqdm( revenues.index[1:-1] ): # <---- Notice tqdm here.
    if revenues.loc[index] > revenues.loc[index-1] and \
        revenues.loc[index] > revenues.loc[index+1]:
        results.append( index )
rev_quarters.iloc[results,:]

```

While the computation is running, a progress bar shows up in the notebook, filling as the computation progresses. It looks like the following example.

32%  323/1000 [00:12<00:27, 25.02it/s]

The numbers indicate that over 300 of the 1000 steps in that large loop are complete, and they have taken 12 seconds (written 00:12) and there are about 27 seconds left (00:27). The loop completes about 25.02 iterations per second. With a progress bar like this, even for a computation that might run for hours, you can tell very quickly how long you will have to wait, and whether it's worth it to wait or if you need to speed up your loop instead.

1.11.7 When the bottleneck is the dataset

Sometimes, you can't get around the fact that you just have to process a lot of data, and that can be slow. Unless you're working for a company that will provide you with some powerful computing resources in the cloud on which to run your Jupyter notebook, so that it runs faster than it does on your laptop (or the free Colab/Deepnote machines), you'll just have to run the slow code. But there are still some ways to make this better.

Don't run it more than you have to. Often, the slow code is something that happens early your work, such as cleaning a huge dataset or searching through it for just the rows you need for your analysis. Once you've written code that does this, save the result to a file with `pd.to_csv()` or `pd.to_pickle()` and don't run that code again.

Don't fall into the trap of thinking that all your code needs to be in one Python script or one Jupyter notebook. If that slow code that cleaned your data never needs to be run again, then once you've run it and saved the output, save the script/notebook, close it, and start a new script or notebook to contain your data analysis code. Then when you re-run your analysis, you don't have to sit around and wait for the data cleaning to happen all over again!

This advice is especially important if the slow part of your work requires fetching data from the Internet. Network downloads are the slowest and least predictable part of your work. Once it's been done correctly, don't run it again.

Do your work on a small dataset. If the dataset you have to analyze is still large enough that your analysis code itself runs slowly as well, try the following. Near the top of your file, replace the actual data with a small sample of it, perhaps using code like this.

```

patients = patients.sample( 3 )
patients

```

	id	height	dose
5	100618	170.18	2.0
3	83	173.00	2.0
0	100615	182.88	2.0

Now the entire rest of my script or notebook will operate on only this tiny DataFrame. (Obviously, you'd want to choose a number larger than three in your code! I'm doing a tiny example here. You might reduce 100,000 rows to just 1,000, for example.)

Then as you create your data analysis code, which inevitably involves running it many times, you won't have to wait for it to process all 100,000 rows of the data. It can work on just 1,000 and run 100x faster. When your analysis code works

and you're ready to write your report, delete the code that creates a small sample of the data and re-run your notebook from the start, now operating on the whole dataset. It will be slower, but you have to sit through that only once.

Danger! Don't forget to delete that cell when your code is polished and you want to do the real, final analysis! I suggest adding a note in giant text at the end of your notebook saying something like, "Don't forget, before you turn this in, USE THE WHOLE DATASET!" Then you'll remember to do that key step before you complete the project.

If the dataset is truly huge, so large that it can't be stored in your computer's memory all at once, then trying to load it will either generate out-of-memory errors or it will slow the process down enormously while the computer tries to use its hard drive as temporary extra memory storage. In such cases, don't forget the tip at the end of [this DataCamp chapter](#) about the `chunksize` parameter. It lets you process large files in smaller chunks.

1.12 Concatenating and Merging DataFrames

See also the slides that summarize a portion of this content.

Warning: This chapter has not yet been written. Check back later.

1.13 Miscellaneous Munging Methods (ETL)

See also the slides that summarize a portion of this content.

Warning: This chapter has not yet been written. Check back later.

1.14 Dashboards

See also the slides that summarize a portion of this content.

Warning: This chapter has not yet been written. Check back later.

1.15 Relations as Graphs - Network Analysis

See also the slides that summarize a portion of this content.

Warning: This chapter has not yet been written. Check back later.

1.16 Relations as Matrices

See also the slides that summarize a portion of this content.

Warning: This chapter has not yet been written. Check back later.

1.17 Introduction to Machine Learning

See also the slides that summarize a portion of this content.

Warning: This chapter has not yet been written. Check back later.

1.18 Big Cheat Sheet

This file summarizes all the coding concepts learned from DataCamp in MA346, as well as those learned in CS230 that remain important in MA346. It is broken into sections in the order in which we encounter the topics in the course, and the course schedule on *the main page* links to each section from the day on which it's learned.

1.18.1 Before Week 2: Review of CS230

Introduction to Python (optional, basic review)

Chapter 1: Python Basics

Comments, which are not executed:

```
# Start with a hash, then explain your code.
```

Print simple data:

```
print( 1 + 5 )
```

Storing data in a variable:

```
num_friends = 1000
```

Integers and real numbers ("floating point"):

```
0, 20, -3192, 16.51309, 0.003
```

Strings:

```
"You can use double quotes."  
'You can use single quotes.'  
'Don\'t forget backslashes when needed.'
```

Booleans:

```
True, False
```

Asking Python for the type of a piece of data:

```
type( 5 ), type( "example" ), type( my_data )
```

Converting among data types:

```
str( 5 ), int( "-120" ), float( "0.5629" )
```

Basic arithmetic (+, −, ×, ÷):

```
1 + 2, 1 - 2, 1 * 2, 1 / 2
```

Exponents, integer division, and remainders:

```
1 ** 2, 1 // 2, 1 % 2
```

Chapter 2: Python Lists

Create a list with square brackets:

```
small_primes = [ 2, 3, 5, 7, 11, 13, 17, 19, 23 ]
```

Lists can mix data of any type, even other lists:

```
# Sublists are name, age, height (in m)
heroes = [ [ 'Harry Potter', 11, 1.3 ],
            [ 'Ron Weasley', 11, 1.5 ],
            [ 'Hermione Granger', 11, 1.4 ] ]
```

Accessing elements from the list is zero-based:

```
small_primes[0]    # == 2
small_primes[-1]   # == 23
```

Slicing lists is left-inclusive, right-exclusive:

```
small_primes[2:4]  # == [5, 7]
small_primes[:4]   # == [2, 3, 5, 7]
small_primes[4:]   # == [11, 13, 17, 19, 23]
```

It can even use a “stride” to count by something other than one:

```
small_primes[0:7:2]    # selects items 0, 2, 4, 6
small_primes[::3]      # selects items 0, 3, 6
small_primes[::-1]     # selects all, but in reverse
```

If indexing gives you a list, you can index again:

```
heroes[1][0]          # == 'Ron Weasley'
```

Modify an item in a list, or a slice all at once:


```
some_list[5] = 10
some_list[5:10] = [ 'my', 'new', 'entries' ]
```

Adding or removing entries from a list:

```
small_primes += [ 27, 29, 31 ]
small_primes = small_primes + [ 37, 41 ]
small_primes.append( 43 ) # to add just one entry
del( heroes[0] ) # Voldemort's goal
del( heroes[:] ) # or, even better, this
```

Copying or not copying lists:

```
# L will refer to the same list in memory as heroes:
L = heroes
# M will refer to a full copy of the heroes array:
M = heroes[:]
```

Chapter 3: Functions and Packages

Calling a function and saving the result:

```
lastSmallPrime = max( small_primes )
```

Getting help on a function:

```
help( max )
```

Methods are functions that belong to an object. (In Python, every piece of data is an object.)

Examples:

```
name = 'jerry'
name.capitalize() # == 'Jerry'
name.count( 'r' ) # == 2
flavors = [ 'vanilla', 'chocolate', 'strawberry' ]
flavors.index( 'chocolate' ) # == 1
```

Installing a package from conda:

```
conda install package_name
```

Ensuring conda forge packages are available:

```
conda config --add channels conda-forge
```

Installing a package from pip:

```
pip3 install package_name
```

Importing a package and using its contents:

```
import math
print( math.pi )
# or if you'll use it a lot and want to be brief:
```

(continues on next page)

(continued from previous page)

```
import math as M
print( M.pi )
```

Importing just some functions from a package:

```
from math import pi, degrees
print( "The value of pi in degrees is:" )
print( degrees( pi ) )          # == 180.0
```

Chapter 4: NumPy

Creating NumPy arrays from Python lists:

```
import numpy as np
a = np.array( [ 5, 10, 6, 3, 9 ] )
```

Elementise computations are supported:

```
a * 2          # == [ 10, 20, 12, 6, 18 ]
a < 10         # == [ True, False, True, True, True ]
```

Use comparisons to subset/select:

```
a[a < 10]      # == [ 5, 6, 3, 9 ]
```

Note: NumPy arrays don't permit mixing data types:

```
np.array( [ 1, "hi" ] )  # converts all to strings
```

NumPy arrays can be 2d, 3d, etc.:

```
a = np.array( [ [ 1, 2, 3, 4 ],
                 [ 5, 6, 7, 8 ] ] )
a.shape      # == (2,4)
```

You can index/select with comma notation:

```
a[1,3]        # == 8
a[0:2,0:2]    # == [[1,2],[5,6]]
a[:,2]        # == [3,7]
a[0,:]        # == [1,2,3,4]
```

Fast NumPy versions of Python functions, and some new ones:

```
np.sum( a )
np.sort( a )
np.mean( a )
np.median( a )
np.std( a )
# and others
```

Python Data Science Toolbox, Part 1 (optional, basic review)

Chapter 1: Writing your own functions

Tuples are like lists, but use parentheses, and are immutable.

```
t = ( 6, 1, 7 )      # create a tuple
t[0]                 # == 6
a, b, c = t           # a==6, b==1, c==7
```

Syntax for defining a function:

(A function that modifies any global variables needs the Python `global` keyword inside to identify those variables.)

```
def function_name ( arguments ):
    """Write a docstring describing the function."""
    # do some things here.
    # note the indentation!
    # and optionally:
    return some_value
    # to return multiple values: return v1, v2
```

Syntax for calling a function:

(Note the distinction between “arguments” and “parameters.”)

```
# if you do not care about a return value:
function_name( parameters )
# if you wish to store the return value:
my_variable = function_name( parameters )
# if the function returns multiple values:
var1, var2 = function_name( parameters )
```

Chapter 2: Default arguments, variable-length arguments, and scope

Defining nested functions:

```
def multiply_by ( x ):
    """Creates a function that multiplies by x"""
    def result ( y ):
        """Multiplies x by y"""
        return x * y
    return result
# example usage:
df["height_in_inches"].apply(
    multiply_by( 2.54 ) ) # result is now in cm
```

Providing default values for arguments:

```
def rand_between ( a=0, b=1 ):
    """Gives a random float between a and b"""
    return np.random.rand() * ( b - a ) + a
```

Accepting any number of arguments:

```
def commas_between ( *args ) :
    """Returns the args as a string with commas"""
    result = ""
    for item in args:
        result += ", " + str(item)
    return result[2:]
commas_between(1,"hi",7)      # == "1,hi,7"
```

Accepting a dictionary of arguments:

```
def inverted ( **kwargs ) :
    """Interchanges keys and values in a dict"""
    result = {}
    for key, value in kwargs.items():
        result[value] = key
    return result
inverted( jim=42, angie=9 )
# == { 42 : 'jim', 9 : 'angie' }
```

Chapter 3: Lambda functions and error handling

Anonymous functions:

```
lambda arg1, arg2: return_value_here
# example:
lambda k: k % 2 == 0      # detects whether k is even
```

Some examples in which anonymous functions are useful:

```
list( map( lambda k: k%2==0, [1,2,3,4,5] ) )
# == [False,True,False,True,False]
list( filter( lambda k: k%2==0, [1,2,3,4,5] ) )
# == [2,4]
reduce( lambda x, y: x*y, [1,2,3,4,5] )
# == 120 (1*2*3*4*5)
```

Raising errors if users call your functions incorrectly:

```
# You can detect problems in advance:
def factorial ( n ) :
    if type( n ) != int:
        raise TypeError( "n must be an int" )
    if n < 0:
        raise ValueError( "n must be nonnegative" )
    return reduce( lambda x,y: x*y, range( 2, n+1 ) )

# Or you can let Python detect them:
def solve_equation ( a, b ) :
    """Solves a*x+b=0 for x"""
    try:
        return -b / a
    except:
        return None
solve_equation( 2, -1 )      # == 0.5
solve_equation( 0, 5 )      # == None
```

Intermediate Python (required review)

Chapter 1: Matplotlib

Conventional way to import matplotlib:

```
import matplotlib.pyplot as plt
```

Creating a line plot:

```
plt.plot( x_data, y_data )      # create plot
plt.show()                     # display plot
```

Creating a scatter plot:

```
plt.scatter( x_data, y_data )  # create plot
plt.show()                     # display plot
# or this alternative form:
plt.plot( x_data, y_data, kind='scatter' )
plt.show()
```

Labeling axes and adding title:

```
plt.xlabel( 'x axis label here' )
plt.ylabel( 'y axis label here' )
plt.title( 'Title of Plot' )
```

Chapter 2: Dictionaries & Pandas

Creating a dictionary directly:

```
days_in_month = {
    "january" : 31,
    "february" : 28,
    "march" : 31,
    "april" : 30,
    # and so on, until...
    "december" : 31
}
```

Getting and using keys:

```
days_in_month.keys()      # == ["january",
                             #     "february",...]
days_in_month["april"]    # == 30
```

Updating dictionary and checking membership:

```
days_in_month["february"] = 29  # update for 2020
"tuesday" in days_in_month       # == False
days_in_month["tuesday"] = 9    # a mistake
"tuesday" in days_in_month       # == True
del( days_in_month["tuesday"] )  # delete mistake
"tuesday" in days_in_month       # == False
```

Build manually from dictionary:

```
import pandas as pd
df = pd.DataFrame( {
    "column label 1": [
        "this example uses...",
        "string data here."
    ],
    "column label 2": [
        100.65, # and numerical data
        -92.04 # here, for example
    ]
    # and more columns if needed
} )
df.index = [
    "put your...",
    "row labels here."
]
```

Import from CSV file:

```
# if row and column headers are in first row/column:
df = pd.read_csv( "/path/to/file.csv",
                  index_col = 0 )
# if no row headers:
df = pd.read_csv( "/path/to/file.csv" )
```

Indexing and selecting data:

```
df["column name"] # is a "Series" (labeled column)
df["column name"].values()
# extract just its values
df[["column name"]] # is a 1-column dataframe
df[["col1", "col2"]] # is a 2-column dataframe
df[n:m] # slice of rows, a dataframe
df.loc["row name"] # is a "Series" (labeled column)
# yes, the row becomes a column
df.loc[["row name"]] # 1-row dataframe
df.loc[["r1", "r2", "r3"]]
# 3-row dataframe
df.loc[["r1", "r2", "r3"], :]
# same as previous
df.loc[:, ["c1", "c2", "c3"]]
# 3-column dataframe
df.loc[["r1", "r2", "r3"], ["c1", "c2"]]
# 3x2 slice of the dataframe
df.iloc[[5]] # is a "Series" (labeled column)
# contains the 6th row's data
df.iloc[[5, 6, 7]] # 3-row dataframe (6th-8th)
df.iloc[[5, 6, 7], :] # same as previous
df.iloc[:, [0, 4]] # 2-column dataframe
df.iloc[[5, 6, 7], [0, 4]]
# 3x2 slice of the dataframe
```

Chapter 3: Logic, Control Flow, and Filtering

Python relations work on NumPy arrays and Pandas Series:

```
<, <=, >, >=, ==, !=
```

Logical operators can combine the above relations:

```
and, or, not      # use these on booleans
np.logical_and(x,y)  # use these on numpy arrays
np.logical_or(x,y)   # (assuming you have imported
np.logical_not(x)    # numpy as np)
```

Filtering Pandas DataFrames:

```
series = df["column"]
filter = series > some_number
df[filter] # new dataframe, a subset of the rows
# or all at once:
df[df["column"] > some_number]
# combining multiple conditions:
df[np.logical_and( df["population"] > 5000,
                  df["area"] < 1250 )]
```

Conditional statements:

```
# Take an action if a condition is true:
if put_condition_here:
    take_an_action()
# Take a different action if the condition is false:
if put_condition_here:
    take_an_action()
else:
    do_this_instead()
# Consider multiple conditions:
if put_condition_here:
    take_an_action()
elif other_condition_here:
    do_this_instead()
elif yet_another_condition:
    do_this_instead2()
else:
    finally_this()
```

Chapter 4: Loops

Looping constructs:

```
while some_condition:
    do_this_repeatedly()
    # as many lines of code here as you like.
    # note that indentation is crucial!
    # be sure to work towards some_condition
    # becoming false eventually!
```

(continues on next page)

(continued from previous page)

```

for item in my_list:
    do_something_with( item )

for index, item in enumerate( my_list ):
    print( "item " + str(index) +
          " is " + str(item) )

for key, value in my_dict.items():
    print( "key " + str(key) +
          " has value " + str(value) )

for item in my_numpy_array:
    # works if the array is one-dimensional
    print( item )

for item in np.nditer( my_numpy_array ):
    # if it is 2d, 3d, or more
    print( item )

for column_name in my_dataframe:
    work_with( my_dataframe[column_name] )

for row_name, row in my_dataframe.iterrows():
    print( "row " + str(row_name) +
          " has these entries: " + str(row) )

# in dataframes, sometimes you can skip the for loop:
my_dataframe["column"].apply( function ) # a Series

```

pandas Foundations (required review)

Chapter 1: Data ingestion & inspection

Basic DataFrame/Series tools:

```

df.head(5)           # first five rows
df.tail(5)           # last five rows
series.head(5)       # head, tail also work on series
df.info()            # summary of the data types used

```

Adding details to reading DataFrames from CSV files:

```

# if no column headers:
df = pd.read_csv( "/path/to/file.csv",
                  index_col = 0, header = None,
                  names = ['column', 'names', 'here'] )

# if any missing data you want to mark as NaN:
# (na_values can be a list of patterns,
# or a dict mapping column names to patterns/lists)
df = pd.read_csv( "/path/to/file.csv",
                  na_values = 'pattern to replace' )

# and many other options! (see the documentation)

```

To get a DataFrame with a date/time index:


```
# read as dates any columns that pandas can:
df = pd.read_csv( "/path/to/file.csv",
                  parse_dates = True )
# read as dates just the columns you specify:
df = pd.read_csv( "/path/to/file.csv",
                  parse_dates = ['column','names'] )
# to use one of those columns as a date/time index:
df = pd.read_csv( "/path/to/file.csv",
                  parse_dates = True,
                  index_col = 'Date' )
# combine multiple columns to form a date:
df = pd.read_csv( "/path/to/file.csv",
                  parse_dates = [[column,indices]] )
```

Export to CSV or XLSX file:

```
df.to_csv( "/path/to/output_file.csv" )
df.to_excel( "/path/to/output_file.xlsx" )
```

You can also create a plot from a Series or dataframe:

```
df.plot()          # or series.plot()
plt.show()
# or to show each column in a subplot:
df.plot( subplots = True )
plt.show()
# or to plot certain columns:
df.plot( x='col name', y='other col name' )
plt.show()
```

A few small ways to customize plots:

```
plt.xscale( 'log' )
plt.yticks( [ 0, 5, 10, 20 ] )
plt.grid()
```

To create a histogram:

```
plt.hist( data, bins=10 )      # 10 is the default
plt.show()
```

To “clean up” so you can start a new plot:

```
plt.clf()
```

Write text onto a plot:

```
plt.text( x, y, 'Text to write' )
```

To save a plot to a file:

```
# before plt.show(), call:
plt.savefig( 'filename.png' ) # or .jpg or .pdf
```

Manipulating DataFrames with pandas (required review)

Chapter 1: Extracting and transforming data

(This builds on the DataCamp Intermediate Python section.)

```
df.iloc[5:7,0:4]      # select ranges of rows/columns
df.iloc[:,0:4]        # select a range, all rows
df.iloc[[5,6],:]      # select a range, all columns
df.iloc[5:,: ]        # all but the first five rows
df.loc['A':'B',:]      # colons can take row names too
                        # (but include both endpoints)
df.loc[:, 'C':'D']     # ...also column names
df.loc['D':'A':-1]     # rows by name, reverse order
```

(This builds on the DataCamp Intermediate Python section.)

```
# avoid using np.logical_and with & instead:
df[(df["population"] > 5000)
   & (df["area"] < 1250 )]
# avoid using np.logical_or with | instead:
df[(df["population"] > 5000)
   | (df["area"] < 1250 )]
# filtering for missing values:
df.loc[:,df.all()]     # only columns with no zeroes
df.loc[:,df.any()]     # only columns with some nonzero
df.loc[:,df.isnull().any()]
                        # only columns with a NaN entry
df.loc[:,df.notnull().all()]
                        # only columns with no NaNs
df.dropna( how='any' )
                        # remove rows with any NaNs
df.dropna( how='all' )
                        # remove rows with all NaNs
```

You can filter one column based on another using these tools.

Apply a function to each value, returning a new DataFrame:

```
def example ( x ):
    return x + 1
df.apply( example )    # adds 1 to everything
df.apply( lambda x: x + 1 )  # same
# some functions are built-in:
df.floordiv( 10 )
# many operators automatically repeat:
df['total pay'] = df['salary'] + df['bonus']
# to extend a dataframe with a new column:
df['new col'] = df['old col'].apply( f )
# slightly different syntax for the index:
df.index = df.index.map( f )
```

You can also map columns through dicts, not just functions.

1.18.2 Before Week 3

Manipulating DataFrames with pandas

Chapter 2: Advanced indexing

Creating a Series:

```
s = pd.Series( [ 5.0, 3.2, 1.9 ] )    # just data
s = pd.Series( [ 5.0, 3.2, 1.9 ],    # data with...
               index = [ 'Mon', 'Tue', 'Wed' ] ) # ...an index
s.index[2:]                          # sliceable
s.index.name = 'Day of Week'         # index name
```

Column headings are also a series:

```
df.columns          # is a pd.Series
df.columns.name     # usually a string
df.columns.values   # column names array
```

Using an existing column as the index:

```
df.index = df['column name'] # once it's the index,
del df['column name']        # it can be deleted
```

Making an index from multiple columns that, when taken together, uniquely identify rows:

```
df = df.set_index( [ 'last_name', 'first_name' ] )
df.index.name     # will be None
df.index.names    # list of strings
df = df.sort_index() # hierarchical sort
df.loc[('Jones', 'Heide')] # index rows by tuples
df.loc[('Jones', 'Heide'), # and you can fetch an
       'birth_date']      # entry that way, too
df.loc['Jones']      # all rows of Joneses
df.loc['Jones':'Menendez'] # many last names
df.loc[(['Jones', 'Wu'], 'Heide'), :]
    # get both rows: Heide Jones and Heide Wu
    # (yes, the colon is necessary for rows)
df.loc[(['Jones', 'Wu'], 'Heide'), 'birth_date']
    # get Heide Jones's and Heide Wu's birth dates
df.loc[('Jones', ['Heide', 'Henry']), :]
    # get full rows for Heide and Henry Jones
df.loc[('Jones', slice('Heide', 'Henry')), :]
    # 'Heide':'Henry' doesn't work inside tuples
```

Chapter 3: Rearranging and reshaping data

If columns A and B together uniquely identify entries in column C, you can create a new DataFrame showing this:

```
new_df = df.pivot( index    = 'A',
                   columns = 'B',
                   values  = 'C' )
# or do this for all columns at once,
# creating a hierarchical column index:
new_df = df.pivot( index    = 'A',
                   columns  = 'B' )
```

You can also invert pivoting, which is called “melting:”

```
old_df = pd.melt( new_df,
                  id_vars = [ 'A' ],           # old index
                  value_vars = [ 'values', 'of', 'column', 'B' ],
                  # optional...pandas can often infer it
                  var_name = 'B',             # these two lines just
                  value_name = 'C' )         # restore column names
```

Convert hierarchical row index to a hierarchical column index:

```
# assume df.index.names is ['A', 'B', 'C']
df = df.unstack( level = 'B' ) # or A or C
# equivalently:
df = df.unstack( level = 1 )   # or 0 or 2
# and this can be inverted:
df = df.stack( level = 'B' )   # for example
```

To change the nesting order of a hierarchical index:

```
df = df.swaplevel( levelindex1, levelindex2 )
df = sort_index() # necessary now
```

If the pivot column(s) aren’t a unique index, use `pivot_table` instead, often with an aggregation function:

```
new_df = df.pivot_table( # this pivot table
                        index = 'A', # is a frequency
                        columns = 'B', # table, because
                        values = 'C', # aggfunc is count
                        aggfunc = 'count' ) # (default: mean)
# other aggfuncs: 'sum', plus many functions in
# numpy, such as np.min, np.max, np.median, etc.
# You can also add column totals at the bottom:
new_df = df.pivot_table(
    index = 'A',
    columns = 'B',
    values = 'C',
    margins = True ) # add column sums
```

Chapter 4: Grouping data

Group all columns except column A by the unique values in column A, then apply some aggregation method to each group:

```
# example: total number of rows for each weekday
df.groupby( 'weekday' ).count()
# example: total sales in each city
df.groupby( 'city' )['sales'].sum()
# multiple column names gives a multi-level index
df.groupby( [ 'city', 'state' ] ).mean()
# you can group by any series with the same index;
# here is an example:
series = df['column A'].apply( np.round )
df.groupby( series )['column B'].sum()
```

The agg method lets us do even more:

```
# you can do multiple aggregations at once;
# this, too, gives a multi-level index:
df.groupby( 'weekday' ).agg( [ 'max', 'sum' ] )
# or you can pass a user-defined function:
def sum_of_squares ( series ):
    return ( series * series ).sum()
df.groupby( 'weekday' )['column name']
    .agg( sum_of_squares )
# or dictionaries can let us apply different
# aggregations to different columns:
df.groupby( 'weekday' )[['Quantity Ordered',
                        'Total Cost']]
    .agg( { 'Quantity Ordered' : 'median',
            'Total Cost'       : 'sum' } )
```

transform is just like apply, except that it must convert each value into exactly one other, thus preserving shape.

```
# example: convert values to zscores
from scipy.stats import zscore
df.groupby( 'region' )['gdp'].transform( zscore )
    .agg( [ 'min', 'max' ] )
# example: impute missing values as medians
def impute_median(series):
    return series.fillna(series.median())
grouped = df.groupby( [ 'col B', 'col C' ] )
df['col A'] = grouped['col A']
    .transform( impute_median )
```

1.18.3 Before Week 4: Review of Visualization in CS230

pandas Foundations

Chapter 2: Exploratory data analysis

Plots from DataFrames:

```
# any of these can be followed with plt.title(),
# plt.xlabel(), etc., then plt.show() at the end:
df.plot( x='col name', y='col name', kind='scatter' )
df.plot( y='col name', kind='box' )
df.plot( y='col name', kind='hist' )
df.plot( kind='box' ) # all columns side-by-side
df.plot( kind='hist' ) # all columns on same axes
```

Histogram options: bins, range, normed, cumulative, and more.

```
df.describe()          # summary statistics
# df.describe() makes calls to df.mean(), df.std(),
# df.median(), df.quantile(), etc...
```

1.18.4 Before Week 5

Intermediate Python

Chapter 5: Case Study: Hacker Statistics

Uniform random numbers from NumPy:

```
np.random.seed( my_int ) # choose a random sequence
# (seeds are optional, but ensure reproducibility)
np.random.rand()         # uniform random in [0,1)
np.random.randint(a,b)   # uniform random in a:b
```

Statistical Thinking in Python, Part 1

Chapter 1: Graphical Exploratory Data Analysis

Plotting a histogram of your data:

```
import matplotlib.pyplot as plt
plt.hist( df['column of interest'] )
plt.xlabel( 'column name (units)' )
plt.ylabel( 'number of [fill in]' )
plt.show()
```

To change the y axis to probabilities:

```
plt.hist( df['column of interest'], normed=True )
```

Sometimes there is a sensible choice of where to place bin boundaries, based on the meaning of the x axis. Example:

```
plt.hist( df['column of percentages'],
          bins=[0,10,20,30,40,50,60,70,80,90,100] )
```

Change default plot styling to Seaborn:

```
import seaborn as sns
sns.set()
# then do plotting afterwards
```

If your data has observations as rows and features as columns, with two features of interest in columns A and B, you can create a “bee swarm plot” as follows.

```
# assuming your dataframe is called df:
sns.swarmplot( x='A', y='B', data=df )
plt.xlabel( 'explain column A' )
plt.ylabel( 'explain column B' )
plt.show()
```

To show a data’s distribution as an Empirical Cumulative Distribution Function plot:

```
# the data must be sorted from lowest to highest:
x = np.sort( df['column of interest'] )
# the y values must count evenly from 0% to 100%:
y = np.arange( 1, len(x)+1 ) / len(x)
# then create and show the plot:
plt.plot( x, y, marker='.', linestyle='none' )
plt.xlabel( 'explain column of interest' )
plt.ylabel( 'ECDF' )
plt.margins( 0.02 ) # 2% margin all around
plt.show()
```

Multiple ECDFs on one plot:

```
# prepare the data as before, but now repeatedly:
# (this could be abstracted into a function)
x = np.sort( df['column 1'] )
y = np.arange( 1, len(x)+1 ) / len(x)
plt.plot( x, y, marker='.', linestyle='none' )
x = np.sort( df['column 2'] )
y = np.arange( 1, len(x)+1 ) / len(x)
# and so on, if there were other columns to plot
plt.plot( x, y, marker='.', linestyle='none' )
# and so on if there are more data series
plt.legend( ('explain x1', 'explain x2'),
            loc='lower right')
# then label axes and show plot as usual (not shown)
```

Chapter 2: Quantitative Exploratory Data Analysis

The mean is the center of mass of the data:

```
np.mean( df['column name'] )  
np.mean( series )
```

The median is the 50th percentile, or midpoint of the data:

```
np.median( df['column name'] )  
np.median( series )
```

Or you can compute any percentile:

```
quartiles = np.percentile(  
    df['column name'], [ 25, 50, 75 ] )  
iqr = quartiles[2] - quartiles[0]
```

Box plots show the quartiles, the IQR, and the outliers:

```
sns.boxplot( x='A', y='B', data=df )  
# then label axes and show plot as above
```

Variance measures the spread of the data, the average squared distance from the mean. Standard deviation is its square root.

```
np.var( df['column name'] )   # or any series  
np.std( df['column name'] )  # or any series
```

Covariance measures correlation between two data series.

```
# get a covariance matrix on of these ways:  
M = np.cov( df['column 1'], df['column 2'] )  
M = np.cov( series1, series2 )  
# extract the value you care about, for example:  
covariance = M[0,1]
```

The Pearson correlation coefficient normalizes this to $[-1, 1]$:

```
# same as covariance, but using np.corrcoef instead:  
np.corrcoef( series1, series2 )
```

Chapter 3: Thinking probabilistically—Discrete variables

Recall these random number generation basics:

```
np.random.seed( my_int )  
np.random.random()      # uniform random in [0,1)  
np.random.randint(a,b)  # uniform random in a:b
```

Sampling many times from some distribution:

```
# if the distribution is built into numpy:  
results = np.random.random( size=1000 )  
# if the distribution is not built into numpy:
```

(continues on next page)

(continued from previous page)

```
simulation_size = 1000    # or any number
results = np.empty( simulation_size )
for i in range( simulation_size ):
    # generate a random number here, however you
    # need to; here is a random example:
    value = 1 - np.random.random() ** 2
    # store it in the list of results:
    results[i] = value
```

Bernoulli trials with probability p :

```
success = np.random.random() < p    # one trial
num_successes = np.random.binomial(
    num_trials, p )                  # many trials
# 1000 experiments, each containing 20 trials:
results = np.random.binomial( 20, p, size=1000 )
```

Poisson distribution (size parameter optional):

```
samples = np.random.poisson(
    mean_arrival_rate, size=1000 )
```

Chapter 4: Thinking probabilistically—Continuous variables

Normal (Gaussian) distribution (size parameter optional):

```
samples = np.random.normal( mean, std, size=1000 )
```

Exponential distribution (time between events in a Poisson distribution, size parameter optional again):

```
samples = np.random.exponential( mean_wait, size=10 )
```

You can take an array of numbers generated by simulation and plot it as an ECDF, as covered in the Graphical EDA chapter, earlier in this week.

Introduction to Data Visualization with Python

NOTE: Only Chapters 1 and 3 are required here.

Chapter 1: Customizing Plots

Break a plot into an $n \times m$ grid of subplots as follows:

(This is preferable to `plt.axes`, not covered here.)

```
# create the grid and begin working on subplot #1:
plt.subplot( n, m, 1 )
plt.plot( x, y )          # this will create plot #1
plt.title( '...' )        # title for plot #1
plt.xlabel( '...' )       # ...and any other options
# keep the same grid and now work on subplot #2:
plt.subplot( n, m, 2 )
```

(continues on next page)

(continued from previous page)

```
# any plot commands here for plot 2,  
# continuing for any further subplots, ending with:  
plt.tight_layout()  
plt.show()
```

Tweak the limits on the axes as follows:

```
plt.xlim( [ min, max ] ) # set x axis limits  
plt.ylim( [ min, max ] ) # set y axis limits  
plt.axis( [ xmin, xmax, ymin, ymax ] ) # both
```

To add a legend to a plot:

```
# when plotting series, give each a label,  
# which will identify it in the legend:  
plt.plot( x1, y1, label='first series' )  
plt.plot( x2, y2, label='second series' )  
plt.plot( x3, y3, label='third series' )  
# then add the legend:  
plt.legend( loc='upper right' )  
# then show the plot as usual
```

To annotate a figure:

```
# add text at some point (here, (10,15)):  
plt.annotate( 'text', xy=(10,15) )  
# add text at (10,15) with an arrow to (5,15):  
plt.annotate( 'text', xytext=(10,15), xy=(5,15),  
              arrowprops={ 'color' : 'red' } )
```

Change plot styles globally:

```
plt.style.available # see list of styles  
plt.style.use( 'style' ) # choose one
```

Chapter 3: Statistical plots with Seaborn

Plotting a linear regression line:

```
import seaborn as sns  
sns.lmplot( x='col 1', y='col 2', data=df )
```

Plotting a linear regression line:

```
import seaborn as sns  
sns.lmplot( x='col 1', y='col 2', data=df )  
plt.show()  
# and the corresponding residual plot:  
sns.residplot( x='col 1', y='col 2', data=df,  
               color='red' ) # color optional
```

Plotting a polynomial regression curve of order n :

```
sns.regplot( x='col 1', y='col 2', data=df,
             order=n )
# this will include a scatter plot, but if you've
# already done one, you can omit redoing it:
sns.regplot( x='col 1', y='col 2', data=df,
             order=n, scatter=None )
```

To do multiple regression plots for each value of a categorical variable in column X, distinguished by color:

```
sns.lmplot( x='col 1', y='col 2', data=df,
            hue='column X', palette='Set1' )
# (many other options exist for palette)
```

Now separate plots into columns, rather than all on one plot:

```
sns.lmplot( x='col 1', y='col 2', data=df,
            row='column X' )
sns.lmplot( x='col 1', y='col 2', data=df,
            col='column X' )
```

Strip plots can visualize univariate distributions, especially useful when broken into categories:

```
sns.stripplot( y='data column', x='category column',
               data=df )
# to add jitter to spread data out a bit in x:
sns.stripplot( y='data column', x='category column',
               data=df, size=4, jitter=True )
```

Swarm plots, covered earlier, are very similar, but can also have colors in them to distinguish categorical variables:

```
sns.swarmplot( y='data column', x='category 1',
               hue='category 2', data=df )
# and you can also change the orientation:
sns.swarmplot( y='category 1', x='data column',
               hue='category 2', data=df,
               orient='h' )
```

Violin plots make curves using kernel density estimation:

```
sns.violinplot( y='data column', x='category 1',
                hue='category 2', data=df )
```

Joint plots for visualizing a relationship between two variables:

```
sns.jointplot( x='col 1', y='col 2', data=df )
# and to add smoothing using KDE:
sns.jointplot( x='col 1', y='col 2', data=df,
               kind='kde' )
# other kind options: reg, resid, hex
```

Scatter plots and histograms for all numerical columns in df:

```
sns.pairplot( df ) # no grouping/coloring
sns.pairplot( df, hue='A' ) # color by column A
```

Visualize a covariance matrix with a heatmap:

```
M = np.cov( df[['col 1','col 2','col3']], # or more
            rowvar=False ) # vars are in columns
# (or you can use np.corrcoef to normalize np.cov)
sns.heatmap( M )
```

1.18.5 Before Week 6

Merging DataFrames with pandas

Chapter 1: Preparing data

The glob module is useful:

```
from glob import glob # built-in module
filenames = glob( '*.csv' ) # filename list
data_frames = [ pd.read_csv(f)
                 for f in filenames ] # import all files
```

You can reorder the rows in a DataFrame with `reindex`:

```
# example: if an index of month or day names were
# sorted alphabetically as strings
# rather than chronologically:
ordered_days = [ 'Mon', 'Tue', 'Wed', 'Thu',
                 'Fri', 'Sat', 'Sun' ]
df.reindex( ordered_days )
# use this to make two dataframes with a common
# index agree on their ordering:
df1.reindex( df2.index )
# in case the indices don't perfectly match,
# NaN values will be inserted, which you can drop:
df1.reindex( df2.index ).dropna()
# or for missing rows, fill with earlier ones:
df.reindex( some_series, method="ffill" )
# (there is also a bfill, for back-fill)
```

You can reorder a DataFrame in preparation for reindexing:

```
# sort by index, ascending or descending:
df = df.sort_index()
df = df.sort_index( ascending=False )
# sort by a column, ascending or descending:
df = df.sort_values( 'column name', # required
                    ascending=False ) # optional
```

Chapter 2: Concatenating data

To add one DataFrame onto the end of another:

```
big_df = df1.append( df2 ) # top: df1, bottom: df2
big_s = s1.append( s2 )    # works for Series, too
# This also stacks indices, so you usually want to:
big_df = big_df.reset_index( drop=True )
```

To add many DataFrames or series on top of one another:

```
big_df = pd.concat( [ df1, df2, df3 ] )
                .reset_index( drop=True )
# equivalently:
big_df = pd.concat( [ df1, df2, df3 ],
                    ignore_index=True )
# or add a hierarchical index to disambiguate:
big_df = pd.concat( [ df1, df2, df3 ],
                    keys=['key1','key2','key3'] )
# equivalently:
big_df = pd.concat( { key1 : df1,
                      key2 : df2,
                      key3 : df3 } )
```

If df2 introduces new columns, and you want to form rows based on common indices, concat by columns:

```
big_df = pd.concat( [ df1, df2 ], axis=1 )
# equivalently:
big_df = pd.concat( [ df1, df2 ], axis='columns' )
# these accept keys=[...] also, or a dict to concat
```

By default, concat performs an “outer join,” that is, index sets are unioned. To intersect them (“inner join”) do this:

```
big_df = pd.concat( [ df1, df2 ], axis=1,
                    join='inner' )
# equivalently:
big_df = df1.join( df2, how='inner' )
```

Chapter 3: Merging data

Inner joins on non-index columns are done with merge.

```
# default merges on all columns present
# in both dataframes:
merged = pd.merge( df1, df2 )
# or you can choose your column:
merged = pd.merge( df1, df2, on='colname' )
# or multiple columns:
merged = pd.merge( df1, df2, on=['col1','col2'] )
# if the columns have different names in each df:
merged = pd.merge( df1, df2,
                  left_on='col1', right_on='col2' )
# to specify meaningful suffixes to replace the
# default suffixes _x and _y:
merged = pd.merge( df1, df2,
                  suffixes=['_from_2011','_from_2012'] )
```

(continues on next page)

(continued from previous page)

```
# you can also specify left, right, or outer joins:
merged = pd.merge( df1, df2, how='outer' )
```

We often have to sort after merging (maybe by a date index), for which there is `merge_ordered`. It most often goes with an outer join, so that's its default.

```
# instead of this:
merged = pd.merge( df1, df2, how='outer' )
            .sorted_values( 'colname' )
# do this, which is shorter and faster:
merged = pd.merge_ordered( df1, df2 )
# it accepts same keyword arguments as merge,
# plus fill_method, like so:
merged = pd.merge_ordered( df1, df2,
                           fill_method='ffill' )
```

When dates don't fully match, you can round dates in the right DataFrame up to the nearest date in the left DataFrame:

```
merged = pd.merge_asof( df1, df2 )
```

1.18.6 Before Week 8

Streamlined Data Ingestion with pandas

Chapter 1: Importing Data from Flat Files

Any file whose rows are on separate lines and whose entries are separated by some delimiter can be read with the same `read_csv` function we've already seen.

```
df = pd.read_csv( "my_csv_file.csv" )    # commas
df = pd.read_csv( "my_tabbed_file.tsv",
                  sep="\t" )             # tabs
```

If you only need some of the data, you can save space:

```
# choose just some columns:
df = pd.read_csv( "my_csv_file.csv", usecols=[
    "use", "only", "these", "columns" ] )
# can also give a list of column indices,
# or a function that filters column names

# choose just the first 100 rows:
df1 = pd.read_csv( "my_csv_file.csv", nrows=100 )
# choose just rows 1001 to 1100,
# re-using the column header from df1:
df2 = pd.read_csv( "my_csv_file.csv",
                  nrows=100, skiprows=1000,
                  header=None,          # skipped it
                  names=list(df1) )    # re-use
```

If pandas is guessing a column's data type incorrectly, you can specify it manually:

```
df = pd.read_csv( "my_geographic_data.csv",
                  dtype={"zipcode":str,
                        "isemployed":bool} )
# to correctly handle bool types:
df = pd.read_csv( "my_geographic_data.csv",
                  dtype={"zipcode":str,
                        "isemployed":bool},
                  true_values=["Yes"],
                  no_values=["No"] )
# note: missing values get coded as True!
# (pandas understands True, False, 0, and 1)
```

If some lines in a file are corrupt, you can ask `read_csv` to skip them and just warn you, importing everything else:

```
df = pd.read_csv( "maybe_corrupt_lines.csv",
                  error_bad_lines=False,
                  warn_bad_lines=True )
```

Chapter 2: Importing Data from Excel Files

If the spreadsheet is a single table of data without formatting:

```
df = pd.read_excel( "my_table.xlsx" )
# nrows, skiprows, usecols, work as before, plus:
df = pd.read_excel( "my_table.xlsx",
                    usecols="C:J,L" ) # excel style
```

If a file contains multiple sheets, choose one by name or index:

```
df = pd.read_excel( "my_workbook.xlsx",
                    sheet_name="budget" )
df = pd.read_excel( "my_workbook.xlsx",
                    sheet_name=3 )
# (the default is the first sheet, index 0)
```

Or load all sheets into an ordered dictionary mapping sheet names to DataFrames:

```
dfs = pd.read_excel( "my_workbook.xlsx",
                     sheet_name=None )
```

Advanced methods of date/time parsing:

```
# standard, as seen before:
df = pd.read_excel( "file.xlsx",
                    parse_dates=True )
# just some cols, in standard date/time format:
df = pd.read_excel( "file.xlsx",
                    parse_dates=["col1", "col2"] )
# what if a date/time pair is split over 2 cols?
df = pd.read_excel( "file.xlsx",
                    parse_dates=[
                        "datetime1",
                        ["date2", "time2"]
                    ] )
# what if we want to control column names?
```

(continues on next page)

(continued from previous page)

```
df = pd.read_excel( "file.xlsx",
                    parse_dates={
                        "name1": "datetime1",
                        "name2": ["date2", "time2"]
                    } )
# for nonstandard formats, do post-processing,
# using a strftime format string, like this example:
df["col"] = pd.to_datetime( df["col"],
                            format="%m%d%Y %H:%M:%S" )
```

Chapter 3: Importing Data from Databases

In SQLite, databases are .db files:

```
# prepare to connect to the database:
from sqlalchemy import create_engine
engine = create_engine( "sqlite:///filename.db" )
# fetch a table:
df = pd.read_sql( "table name", engine )
# or run any kind of SQL query:
df = pd.read_sql( "PUT QUERY CODE HERE", engine )
# if the query code is big:
query = """PUT YOUR SQL CODE
          HERE ON AS MANY LINES
          AS YOU LIKE;"""
df = pd.read_sql( query, engine )
# or get a list of tables:
print( engine.table_names() )
```

Chapter 4: Importing JSON Data and Working with APIs

From a file or string:

```
# from a file:
df = pd.read_json( "filename.json" )
# from a string:
df = pd.read_json( string_containing_json )
# can specify dtype, as with read_csv:
df = pd.read_json( "filename.json",
                  dtype={"zipcode": str} )
# also see pandas documentation for JSON "orient":
# records, columns, index, values, or split
```

From the web with an API:

```
import requests
response = requests.get(
    "http://your.api.com/goes/here",
    headers = {
        "dictionary" : "with things like",
        "username" : "or API key"
    },
    params = {
```

(continues on next page)

(continued from previous page)

```

        "dictionary" : "with options as",
        "required by" : "the API docs"
    } )
data = response.json() # ignore metadata
result = pd.DataFrame( data )
# or possibly some part of the data, like:
result = pd.DataFrame( data["some key"] )
# (you must inspect it to know)

```

If the JSON has nested objects, you can flatten:

```

from pandas.io.json import json_normalize
# instead of this line:
result = pd.DataFrame( data["maybe a column"] )
# do this:
result = json_normalize( data["maybe a column"],
                        sep="_" )
# (if there is deep nesting, see the record_path,
# meta, and meta_prefix options)

```

1.18.7 Before Week 9

Introduction to SQL

Chapter 1: Selecting columns

SQL (“sequel”) means Structured Query Language. A SQL database contains tables, each of which is like a DataFrame.

```

-- A single-line SQL comment
/*
A multi-line
SQL comment
*/

```

To fetch one column from a table:

```
SELECT column_name FROM table_name;
```

To fetch multiple columns from a table:

```
SELECT column1, column2 FROM table_name;
SELECT * FROM table_name;    -- all columns

```

To remove duplicates:

```
SELECT DISTINCT column_name
FROM table_name;
```

To count rows:

```
SELECT COUNT(*)
FROM table_name;      -- counts all the rows
SELECT COUNT(column_name)
FROM table_name;      -- counts the non-
                      -- missing values in just that column
SELECT COUNT(DISTINCT column_name)
FROM table_name;      -- # of unique entries
```

If a result is huge, you may want just the first few lines:

```
SELECT column FROM table_name
LIMIT 10;              -- only return 10 rows
```

Chapter 2: Filtering rows

(selecting a subset of the rows using the WHERE keyword)

Using the comparison operators <, >, =, <=, >=, and <>, plus the inclusive range filter BETWEEN:

```
SELECT * FROM table_name
WHERE quantity >= 100;  -- numeric filter
SELECT * FROM table_name
WHERE name = 'Jeff';   -- string filter
```

Using range and set filters:

```
SELECT title, release_year FROM films
WHERE release_year BETWEEN 1990 AND 1999;
                      -- range filter

SELECT * FROM employees
WHERE role IN ('Engineer', 'Sales');
                      -- set filter
```

Finding rows where specific columns have missing values:

```
SELECT * FROM employees
WHERE role IS NULL;
```

Combining filters with AND, OR, and parentheses:

```
SELECT * FROM table_name
WHERE quantity >= 100
  AND name = 'Jeff';   -- one combination
SELECT title, release_year FROM films
WHERE release_year >= 1990
  AND release_year <= 1999
  AND ( language = 'French'
      OR language = 'Spanish' )
  AND gross > 2000000;  -- many
```

Using wildcards (%) and _ to filter strings with LIKE:

```
SELECT * FROM employees
WHERE name LIKE 'Mac%'; -- e.g., MacEwan
SELECT * FROM employees
WHERE id NOT LIKE '%00'; -- e.g., 352800
```

(continues on next page)

(continued from previous page)

```
SELECT * FROM employees
WHERE name LIKE 'D_n'; -- e.g., Dan, Don
```

Chapter 3: Aggregate Functions

We've seen this function before; it is an aggregator:

```
SELECT COUNT(*)
FROM table_name; -- counts all the rows
```

Some other aggregating functions: SUM, AVG, MIN, MAX. The resulting column name is the function name (e.g., MAX).

To give a more descriptive name:

```
SELECT MIN(salary) AS lowest_salary,
       MAX(salary) AS highest_salary
FROM employees;
```

You can also do arithmetic on columns:

```
SELECT budget/1000 AS budget_in_thousands
FROM projects; -- convert a column
SELECT hours_worked * hourly_pay
FROM work_log WHERE date > '2019-09-01';
-- create a column
SELECT count(start_date)*100.0/count(*)
FROM table_name; -- percent not missing
```

Chapter 4: Sorting and grouping

Sorting happens only after selecting:

```
SELECT * FROM employees
ORDER BY name; -- ascending order
SELECT * FROM employees
ORDER BY name DESC; -- descending order
SELECT name,salary FROM employees
ORDER BY role,name; -- multiple columns
```

Grouping happens after selecting but before sorting. It is used when you want to apply an aggregate function like COUNT or AVG not across the whole result set, but to groups within it.

```
-- Compute average salary by role:
SELECT role,AVG(salary) FROM employees
GROUP BY role;
-- How many people are in each division?
-- (sorting results by division name)
SELECT division,COUNT(*) FROM employees
GROUP BY division
ORDER BY division;
```

Every selected column except the one(s) you're aggregating must appear in your GROUP BY.

To filter by a condition (like with WHERE but now applied to each group) use the HAVING keyword:

```
-- Same as above, but omit tiny divisions:
SELECT division, COUNT(*) FROM employees
GROUP BY division
HAVING COUNT(*) >= 10
ORDER BY division;
```

1.18.8 Additional Useful References

Python Data Science Toolbox, Part 2

Chapter 1: Using iterators in PythonLand

To convert an iterable to an iterator and use it:

```
my_iterable = [ 'one', 'two', 'three' ] # example
my_iterator = iter( my_iterable )
first_value = next( my_iterator )      # 'one'
second_value = next( my_iterator )     # 'two'
# and so on
```

To attach indices to the elements of an iterable:

```
my_iterable = [ 'one', 'two', 'three' ] # example
with_indices = enumerate( my_iterable )
my_iterator = iter( with_indices )
first_value = next( my_iterator )       # (0, 'one')
second_value = next( my_iterator )     # (1, 'two')
# and so on; see also "Looping Constructs" earlier
```

To join iterables into tuples, use `zip`:

```
iterable1 = range( 5 )
iterable2 = 'five!'
iterable3 = [ 'How', 'are', 'you', 'today', '?' ]
all = zip( iterable1, iterable2, iterable3 )
next( all )      # (0, 'f', 'How')
next( all )      # (1, 'i', 'are')
# and so on, or use this syntax:
for x, y in zip( iterable1, iterable2 ):
    do_something_with( x, y )
```

Think of `zip` as converting a list of rows into a list of columns, a “matrix transpose,” which is its own inverse:

```
row1 = [ 1, 2, 3 ]
row2 = [ 4, 5, 6 ]
cols = zip( row1, row2 )      # swap rows and columns
print( *cols )                # (1,4) (2,5) (3,6)
cols = zip( row1, row2 )      # restart iterator
undo1, undo2 = zip( *cols )   # swap rows/cols again
print( undo1, undo2 )         # (1,2,3) (4,5,6)
```

Pandas can read CSV files into DataFrames in chunks, creating an iterable out of a file too large for memory:

```
import pandas as pd
for chunk in pd.read_csv( filename, chunksize=100 ):
    process_one_chunk( chunk )
```

Chapter 2: List comprehensions and generators

List comprehensions build a list from an output expression and a `for` clause:

```
[ n**2 for n in range(3,6) ]      # == [9,16,25]
```

You can nest list comprehensions:

```
[ (i,j) for i in range(3) for j in range(4) ]
# == [(0,0), (0,1), (0,2), (0,3),
#      (1,0), (1,1), (1,2), (1,3),
#      (2,0), (2,1), (2,2), (2,3)]
```

You can put conditions on the `for` clause:

```
[ (i,j) for i in range(3) for j in range(3)
    if i + j > 2 ] # == [ (1,2), (2,1), (2,2) ]
```

You can put conditions in the output expression:

```
some_data = [ 0.65, 9.12, -3.1, 2.8, -50.6 ]
[ x if x >= 0 else 'NEG' for x in some_data ]
# == [ 0.65, 9.12, 'NEG', 2.8, 'NEG' ]
```

A dict comprehension creates a dictionary from an output expression in `key:value` form, plus a `for` clause:

```
{ a: a.capitalize() for a in ['one','two','three'] }
# == { 'one':'One', 'two':'Two', 'three':'Three' }
```

Just like list comprehensions, but with parentheses:

```
g = ( n**2 for n in range(3,6) )
next( g )      # == 9
next( g )      # == 16
next( g )      # == 25
```

You can build generators with functions and `yield`:

```
def just_like_range ( a, b ):
    counter = a
    while counter < b:
        yield counter
        counter += 1
list( just_like_range( 5, 9 ) ) # == [5,6,7,8]
```

Introduction to Data Visualization with Python

Chapter 2: Plotting 2D arrays

To plot a bivariate function using colors:

```
# choose the sampling points in both axes:
u = np.linspace( xmin, xmax, num_xpoints )
v = np.linspace( ymin, ymax, num_ypoints )
# create pairs from these axes:
x, y = np.meshgrid( u, v )
# broadcast a function across those points:
z = x**2 - y**2
# plot it in color:
plt.pcolor( x, y, z )
plt.colorbar() # optional but helpful
plt.axis( 'tight' ) # remove whitespace
plt.show()
# optionally, the pcolor call can take a color
# map parameter, one of a host of palettes, e.g.:
plt.pcolor( x, y, z, cmap='autumn' )
```

To make a contour plot instead of a color map plot:

```
# replace the pcolor line with this:
plt.contour( x, y, z )
plt.contour( x, y, z, 50 ) # choose num. contours
plt.contourf( x, y, z ) # fill the contours
```

To make a bivariate histogram:

```
# for rectangular bins:
plt.hist2d( x, y, bins=(xbins,ybins) )
plt.colorbar()
# with optional x and y ranges:
plt.hist2d( x, y, bins=(xbins,ybins),
            range=((xmin,xmax),(ymin,ymax)) )
# for hexagonal bins:
plt.hexbin( x, y,
            gridsize=(num_x_hexes,num_y_hexes) )
# with optional x and y ranges:
plt.hexbin( x, y,
            gridsize=(num_x_hexes,num_y_hexes),
            extent=(xmin,xmax,ymin,ymax) )
```

To display an image from a file:

```
image = plt.imread( 'filename.png' )
plt.imshow( image )
plt.axis( 'off' ) # axes don't apply here
plt.show()
# to collapse a color image to grayscale:
gray_img = image.mean( axis=2 )
plt.imshow( gray_img, cmap='gray' )
# to alter the aspect ratio:
plt.imshow( gray_img, aspect=height/width )
```

1.19 Anaconda Installation

Anaconda is a tool that installs Python together with the `conda` package manager and several related apps, tools, and packages. It's one of the easiest ways to get Python installed on your system and ready to use for data work.

These instructions are written primarily for Windows, with Mac instructions in parentheses.

1.19.1 Visit the Anaconda website

It is at this URL: www.anaconda.com/distribution

It looks like this:

1.19.2 Choose your OS

Scroll down on that same website and click the Windows link to indicate that you want to download the installer for Windows.

(Mac users obviously click the macOS link instead.)

1.19.3 Download the Installer

Click the download button for the Python 3.7 distribution of Anaconda, as shown on the left below.

1.19.4 Run the Installer

Run the installer once it's downloaded, probably by clicking the downloaded file in your browser's list of downloaded files, usually at the bottom left of the window.

(For Mac users, this will be a `.pkg` file instead of an `.exe`.)

Accept all the default choices during installation. This may take up to 10 minutes.

(For Mac users, the installer will look slightly different than the one above.)

After this, you may wish to *install VS Code* as well.

1.20 VS Code for Python Installation

This assumes you have *installed Anaconda* already.

1.20.1 Open the Anaconda Navigator

Start Menu > Anaconda3 > Anaconda Navigator(On Mac: Finder > Applications > Anaconda Navigator.app.)

1.20.2 Find and Install the VS Code application

Scroll down in the list of applications until you find VS Code (Visual Studio Code, by Microsoft).

Click the Install button beneath it.

Once you have installed VS Code, its application icon will change to contain a “Launch” button.

Click that button now to launch VS Code.

1.20.3 Installing and Configuring Code Runner

VS Code is all ready to let you edit Python code, but there’s an Extension to VS Code, called Code Runner, that makes it more convenient to run Python code when testing your work. Let’s install it now.

Click the Extensions button on the left of the VS Code window. It is the bottom button shown below, which looks like four squares:

Then search for the Code Runner Extension as shown below and click its install button. (The install button is green and is just below and to the right of the large orange “.run” icon.)

The Code Runner Extension in the Extensions list will then have a settings gear icon, as shown below.

Click that gear icon to bring up the settings menu for Code Runner, as shown below.

Choose “Configure Extension Settings,” the bottom item on that menu.

It will bring up a settings window as shown below.

Scroll down until you find the settings for saving files before running, and check both boxes, as shown here.

Your Code Runner Extension is correctly configured. Try it out as shown on the next page.

1.20.4 Testing your Installation

Let’s verify now that you can successfully run Python code.

Create a new file:

Save the file and give it the name `test.py` to indicate that it is a Python file.

Enter the following small amount of Python code in the new, empty file.

Be sure to press Enter after the code to start a new line!

Save the file again.

Run the file by clicking the small Run icon (which looks like a “Play” triangle) on the top right of the window. (If you don’t see this button, check your work above—did you save the file with a `.py` extension?)

You should see the following output at the bottom of the window, indicating that your code was run, and produced the output 4 (which is the result of $2+2$, of course).

(The first line means that the “python” command was run on the `test.py` file you saved. The final line means the process ended with error code 0, which means no errors, and took 0.102 seconds in total.)

You have a successful Python installation that you can run from Visual Studio!

1.21 GB213 Review in Python

1.21.1 We're not covering everything

We're omitting basic probability issues like experiments, sample spaces, discrete probabilities, combinations, and permutations. At the end we'll provide links to example topics. But everything else we'll cover at least briefly.

We begin by importing the necessary modules.

```
import numpy as np
import pandas as pd
import scipy.stats as stats
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

1.21.2 Discrete Random Variables

(For continuous random variables, *see further below*.)

Discrete random variables taken on a finite number of different values. For example, a Bernoulli trial is either 0 or 1 (usually meaning failure and success, respectively). You can create random variables using `scipy.stats` as follows.

Creating them

```
b1 = stats.bernoulli( 0.25 ) # probability of success
b2 = stats.binom( 10, 0.5 ) # number of trials, prob. of success on each
```

Computing probabilities from a Discrete Random Variable

```
b1.pmf( 0 ), b1.pmf( 1 ) # stands for "probability mass function"
```

```
(0.75, 0.25)
```

The same code works for any random variable, not just `b1`.

Generating values from a Discrete Random Variable

```
b1.rvs( 10 ) # asks for 10 random values (rvs)
```

```
array([0, 1, 0, 1, 0, 0, 0, 0, 0, 0])
```

The same code works for any random variable, not just `b1`.

Computing statistics about a Discrete Random Variable

```
b1.mean(), b1.var(), b1.std()  # mean, variance, standard deviation
```

```
(0.25, 0.1875, 0.4330127018922193)
```

The same code works for any random variable, not just `b1`.

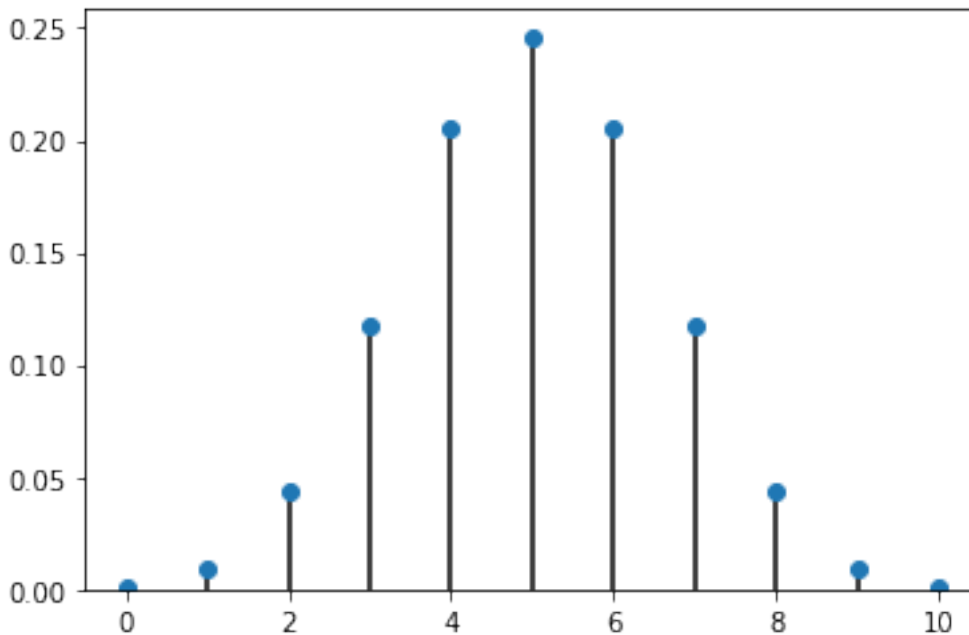
Plotting the Distribution of a Discrete Random Variable

Here's a function you can use to plot (almost) any discrete probability distribution.

```
def plot_discrete_distribution ( rv ) :  
    xmin, xmax = rv.ppf( 0.0001 ), rv.ppf( 0.9999 )  
    x = np.arange( xmin, xmax+1 )  
    y = rv.pmf( x )  
    plt.plot( x, y, 'o' )  
    plt.vlines( x, 0, y )  
    plt.ylim( bottom=0 )
```

Example use:

```
plot_discrete_distribution( b2 )
```



1.21.3 Continuous Random Variables

(For discrete random variables, *see further above*.)

Continuous random variables take on an infinite number of different values, sometimes in a certain range (like the uniform distribution on $[0,1]$, for example) and sometimes over the whole real number line (like the normal distribution, for example).

Creating them

```
# for uniform on the interval [a,b]: use loc=a, scale=b-a
u = stats.uniform( loc=10, scale=2 )
# for normal use loc=mean, scale=standard deviation
n = stats.norm( loc=100, scale=5 )
# for t, same as normal, plus df=degrees of freedom
t = stats.t( df=15, loc=100, scale=5 )
```

Computing probabilities from a Continuous Random Variable

For a continuous random variable, you cannot compute the probability that it will equal a precise number, because such a probability is always zero. But you can compute the probability that the value falls within a certain interval on the number line.

To do so for an interval $[a,b]$, compute the total probability accumulated up to a and subtract it from that up to b , as follows.

```
a, b = 95, 100 # or any values
n.cdf( b ) - n.cdf( a ) # probability of being in that interval
```

```
0.3413447460685429
```

The same code works for any continuous random variable, not just n .

Generating values from a Continuous Random Variable

```
n.rvs( 10 ) # same as for discrete random variables
```

```
array([ 97.9895976 , 103.57790101, 102.14852225, 104.84689142,
        99.02659809, 107.33275204,  95.22139243, 100.98510897,
        105.20215099, 100.80143421])
```

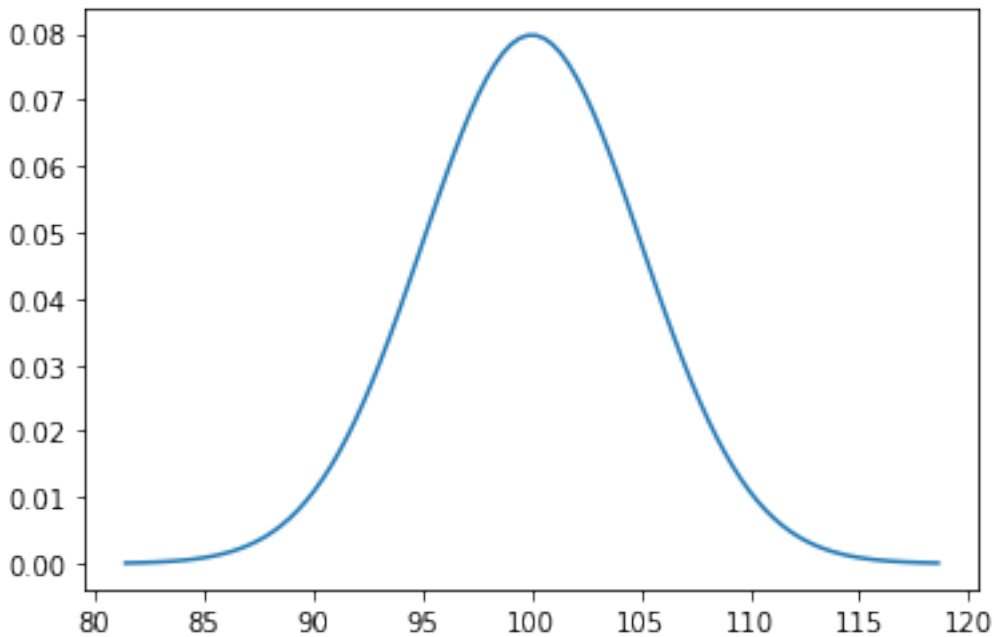
Plotting the Distribution of a Continuous Random Variable

Here's a function you can use to plot the center 99.98% of any continuous probability distribution.

```
def plot_continuous_distribution ( rv ):
    xmin, xmax = rv.ppf( 0.0001 ), rv.ppf( 0.9999 )
    x = np.linspace( xmin, xmax, 100 )
    y = rv.pdf( x )
    plt.plot( x, y )
```

Example use:

```
plot_continuous_distribution( n )
```



1.21.4 Confidence Intervals

Recall from GB213 that certain assumptions about normality must hold in order for you to do statistical inference. We do not cover those here; refer to your GB213 text or notes.

Here we cover a confidence interval for the sample mean using confidence level α , which must be between 0 and 1 (typically 0.95).

```
α = 0.95
# normally you'd have data; for this example, I make some up:
data = [ 435, 542, 435, 4, 54, 43, 5, 43, 543, 5, 432, 43, 36, 7, 876, 65, 5 ]
est_mean = np.mean( data ) # estimate for the population mean
sem = stats.sem( data )    # standard error for the sample mean
# margin of error:
moe = sem * stats.t.ppf( ( 1 + α ) / 2, len( data ) - 1 )
( est_mean - moe, est_mean + moe ) # confidence interval
```

```
(70.29847811072423, 350.0544630657464)
```

1.21.5 Hypothesis Testing

Again, in GB213 you learned what assumptions must hold in order to do a hypothesis test, which I do not review here.

Let H_0 be the null hypothesis, the currently held belief. Let H_a be the alternative, which would result in some change in our beliefs or actions.

We assume some chosen value $0 \leq \alpha \leq 1$, which is the probability of a Type I error (false positive, finding we should reject H_0 when it's actually true).

Two-sided test for $H_0 : \mu = \bar{x}$

Say we have a population whose mean μ is known to be 10. We take a sample x_1, \dots, x_n and compute its mean, \bar{x} . We then ask whether this sample is significantly different from the population at large, that is, is $\mu = \bar{x}$? We can do a two-sided test of $H_0 : \mu = \bar{x}$ as follows.

```
α = 0.05
μ = 10
sample = [ 9, 12, 14, 8, 13 ]
t_statistic, p_value = stats.ttest_1samp( sample, μ )
reject_H0 = p_value < α
α, p_value, reject_H0
```

```
(0.05, 0.35845634462296455, False)
```

The output above says that the data does NOT give us enough information to reject the null hypothesis. So we should continue to assume that the sample is like the population, and $\mu = \bar{x}$.

Two-sided test for $H_0 : \bar{x}_1 = \bar{x}_2$

What if we had wanted to do a test for whether two independent samples had the same mean? We can ask that question as follows. (Here we assume they have equal variances, but you can turn that assumption off with a third parameter to `ttest_ind`.)

```
α = 0.05
sample1 = [ 6, 9, 7, 10, 10, 9 ]
sample2 = [ 12, 14, 10, 17, 9 ]
t_statistics, p_value = stats.ttest_ind( sample1, sample2 )
reject_H0 = p_value < α
α, p_value, reject_H0
```

```
(0.05, 0.02815503832602318, True)
```

The output above says that the two samples DO give us enough information to reject the null hypothesis. So the data suggest that the two samples have different means.

1.21.6 Linear Regression

Creating a linear model of data

Normally you would have data that you wanted to model. But in this example notebook, I have to make up some data first.

```
df = pd.DataFrame( {
    "height" : [ 393, 453, 553, 679, 729, 748, 817 ], # completely made up
    "width"  : [ 24, 25, 27, 36, 55, 68, 84 ] # also totally pretend
} )
```

As with all the content of this document, the assumptions required to make the technique applicable are not covered in detail, but in this case we at least review them briefly. To ensure that linear regression is applicable, one should verify:

1. We have two columns of numerical data of the same length.
2. We have made a scatter plot and observed a seeming linear relationship.
3. We know that there is no autocorrelation.
4. We will check later that the residuals are normally distributed.
5. We will check later that the residuals are homoscedastic.

To create a linear model, use `stats` as follows.

```
model = stats.linregress( df.height, df.width )
model
```

```
LinregressResult(slope=0.1327195637885226, intercept=-37.32141898334582, rvalue=0.
  8949574425541466, pvalue=0.006486043236692156, stderr=0.029588975845594334)
```

A linear model is usually written like so:

$$y = \beta_0 + \beta_1 x$$

The slope is β_1 and the intercept is β_0 .

```
β0 = model.intercept
β1 = model.slope
β0, β1
```

```
(-37.32141898334582, 0.1327195637885226)
```

From the output above, our model would therefore be the following (with some rounding for simplicity):

$$y = -37.32 + 0.132x$$

To know how good it is, we often ask about the R^2 value.

```
R = model.rvalue
R, R**2
```

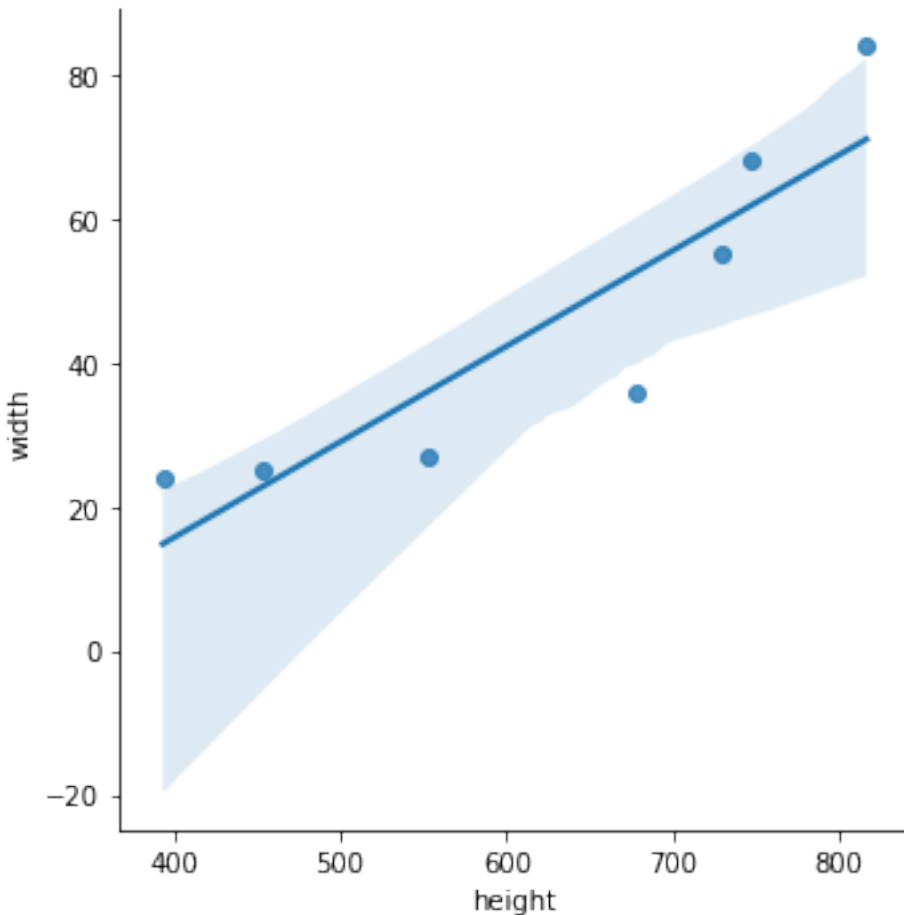
```
(0.8949574425541466, 0.8009488239830586)
```

In this case, R^2 would be approximately 0.895^2 , or about 0.801. Thus our model explains about 80.1% of the variability in the data.

Visualizing the model

The Seaborn visualization package provides a handy tool for making scatterplots with linear models overlaid. The light blue shading is a confidence band we will not cover.

```
sns.lmplot( x='height', y='width', data=df )  
plt.show()
```



1.21.7 Other Topics

ANOVA

Analysis of variance is an optional topic your GB213 class may or may not have covered, depending on scheduling and instructor choices. If you covered it in GB213 and would like to see how to do it in Python, check out [the Scipy documentation for `f_oneway`](#).

χ^2 Tests

Chi-squared (χ^2) tests are another optional GB213 topic that your class may or may not have covered. If you are familiar with it and would like to see how to do it in Python, check out [the Scipy documentation for `chisquare`](#).